14.95

Beyond Games: Systems Software for Your 6502 Personal Computer

Ken Skier

Beyond Games:
Systems Software for Your
6502
Personal Computer

Ken Skier

BYTE/McGraw-Hill, Book Division, 70 Main St, Peterborough NH 03458

Beyond Games: Systems Software for Your 6502 Personal Computer

Library of Congress Cataloging in Publication Data

Skier, Ken.
Beyond games.

Includes index.
1. 6502 (Computer)—Programming. I. Title.
QA76.8.S63S59 001.64'2 80-28512
ISBN 0-07-057860-5

Cover photo by Bob Hamor

Text set in Paladium
by BYTE Publications

Edited by Blaise Liffick

Design and Production Supervision
by Ellen Klempner

Copy Edited by Rich Friedman

Printed and bound using 50#MH Matte

Table of Contents

Introduction

Objectives

Sometimes I hear people talk about how smart computers have become. But computers aren't smart: programmers are. Programmers make microprocessors act like calculators, moon landers, or income tax preparers. Programmers must be smart, because by themselves microprocessors can't do much of anything.

Sound programming, then, is fundamental to successful computer use. With this principle in mind, this book has two objectives: first, to introduce newcomers to some of the techniques, terminology, and power of assembly-language programming in general, and of the 6502 in particular; and second, to present a set of software tools to use in developing assembly-language programs for the 6502.

Chapter 1 takes you on a quick tour of your computer's hardware and software; Chapters 2 thru 4 comprise a short course in assembly-language programming for those readers new to the subject. The rest of the book presents source listings, object code, and assembler listings for programs that you may enter into your computer and run.

Programmers have long sought to develop small and fast programs with the unfortunate result that occasionally code has been written that is unreadable (and even unworkable) simply because a programmer wanted to save a few bytes or a few cycles. In certain instances when memory space is particularly tight or execution time is critical, readability is sacrificed for performance. But today the average programmer is not forced to make this choice. Of course, all other things being equal, I, too, value programs that are quick and compact.

But how often are all other things equal?

While developing the programs that appear in this book, I had a number of objectives, most of them more important than the speed or size of a block of code. I designed these programs to be:

Useful: No program is presented simply to demonstrate a particular programming technique. All of the programs in this book were written because I needed certain things done — usually something I didn't want to be bothered with doing myself. The monitor monitors, the disassembler disassembles, and the text editor lets me enter and edit text strings. These programs earn their keep.

Easy to Use: Simply by glancing at the screen you can tell which program is running and what mode it is in. When a program needs information, it asks you for it and allows you to correct mistakes you might make while answering. This software doesn't require you to remember the addresses of programs or of variables. Functions are mapped to individual keys, and you can assign functions to keys in any way that makes sense to you.

Readable: A beginning 6502 programmer should be able to understand the workings of every program in this book. The labels and comments in the listings were carefully chosen to reveal the purpose of each variable, subroutine, and line of code. I am writing first and foremost for you, the reader, not for the 6502.

Portable: The book's software runs on an Apple II, an Atari 400 or 800, an Ohio Scientific (OSI) Challenger I-P, or a PET 2001. With proper initialization of the System Data Block, it should run on *any* 6502-based computer equipped with a keyboard and a memory-mapped, character-graphics video display.

Compatible: These routines are very good neighbors. As long as the other software in your system does not use the second 4 K bytes of memory (hexadecimal memory locations 1000 thru 1FFF), there should be no conflict between your software and the software in this book. In particular, most of the software in this book preserves the zero page, so your software may use the zero page as much as you like, and you won't be bothered with having to save and restore it before and after calls to the software presented herein.

Expandable: The programs in this book are highly modular, and you may extend or restructure them to meet your individual needs. System-specific subroutines are called indirectly, so that other subroutines may be substituted for them, and most values are treated as variables, rather than as constants hard-wired into the code. There are no monolithic programs in this book; they're all subroutines and may be combined in many ways to build powerful new structures.

Compact: I know that every personal computer has exactly the same available memory: too little. I also know ways to write a program in ten or twenty percent less space. But if doing so required sacrificing readability, portability, or expandability, I did not do so. In many cases I feared that to save a byte, I might lose a reader's clear understanding of how a program works. I considered that too great a price to pay for a somewhat smaller program.

Fast: Assuming that the above objectives have been met, the software in this book has been developed to operate as quickly as possible. But in any trade-off between speed and the other objectives, speed loses. A fast program that you can't understand holds little value. None of the programs in this book are likely to make you complain about how long you have to wait. I can't tell if I'm waiting an extra millisecond. Can you?

So go ahead. Read. Program. Enjoy!

Chapter 1:

Your Computer

The software in this book can run on a number of computers because it assumes very little about the host machine. Let's examine these assumptions and in so doing take a quick tour of your computer.

The 6502 Microprocessor

We'll start with the 6502 microprocessor, the component in your system that actually computes. By itself, the 6502 can't do much. It has three *registers* (special memory areas for storing the data upon which the program is operating), called A, X, and Y, which can each hold a number in the range of 0 to 255. Different registers have different capabilities. For example, if a number is in A (the accumulator), the 6502 can add to it, or subtract from it, any value up to 255. But if a number is in the X register or the Y register, the 6502 can only increment or decrement that number (ie: add or subtract one from it).

The 6502 can also set one register equal to the value of another register, and it can store the contents of any register anywhere in memory, or load any register from any location in memory. Thus, although the 6502 can only operate on one number at a time, it can operate on many numbers, just by loading registers from various locations in memory, operating on the registers, and then storing the results of those operations back into memory.

Types of Memory

You may have heard that a computer stores information as a series of ones and zeros. This is because the computer's memory is simply an elaborate array of switches, and an individual switch can have only two states: closed or open. These two states may also be expressed as on and off, or as one and zero.

Not all memory switches are the same. Some, in what is called ROM (read-only memory), are hard-wired into your computer's circuitry and cannot be changed except by physically replacing the ROM circuits containing those switches. Others, in what is called RAM (random-access memory) or programmable memory, can be changed by the processor. The 6502 can open or close any of the switches, called bits (binary digits), in its programmable memory, and later on read what it "wrote" into that memory. Figure 1.1 shows how the processor has access to read-only memory and programmable memory.

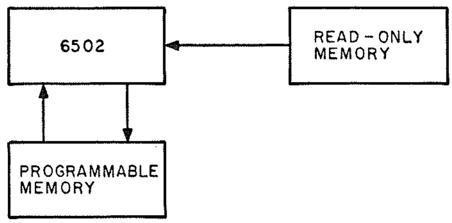

Figure 1.1: *How the 6502 interacts with memory. The arrows indicate the flow of data.*

A third kind of memory is set by some external device, not by the 6502. Such memory switches are called *input ports,* and may be connected to keyboards, terminals, burglar alarms — virtually anything that can generate an electrical signal. The 6502 perceives these externally generated signals by reading the appropriate input ports.

Yet another kind of memory switch, called an *output port,* generates a high or a low voltage on some particular wire depending on whether the 6502 sets a given memory switch to a one or a zero. One or more of these output ports can enable the 6502 to "talk" to the outside world.

Now don't jump up and think I'm going to show you how to synthesize speech in this book. "Talk" is just my way of anthropomorphizing the 6502. It will happen elsewhere in this book, when the 6502 "sees," "remembers," and "knows" what to do. Of course the 6502 doesn't see, remember, or know anything, but I often find it helpful to put myself in its place. That way I can better understand how a program will run, or why a program doesn't run, and I *do* see, remember, and know things.

But don't take such verbs too literally. The 6502 doesn't talk. It causes signals to be generated that may be sensed by other devices, such as cassette recorders, printers, disk drives — and yes, even speech synthesizers. But not in this book.

Some peripheral devices are actually connected to both an input and an output port. Examples of these devices are cassette tape machines and floppy-disk drives, which are mass-storage or secondary-storage devices. Figure 1.2 summarizes the processor's access to memory and to peripheral devices.

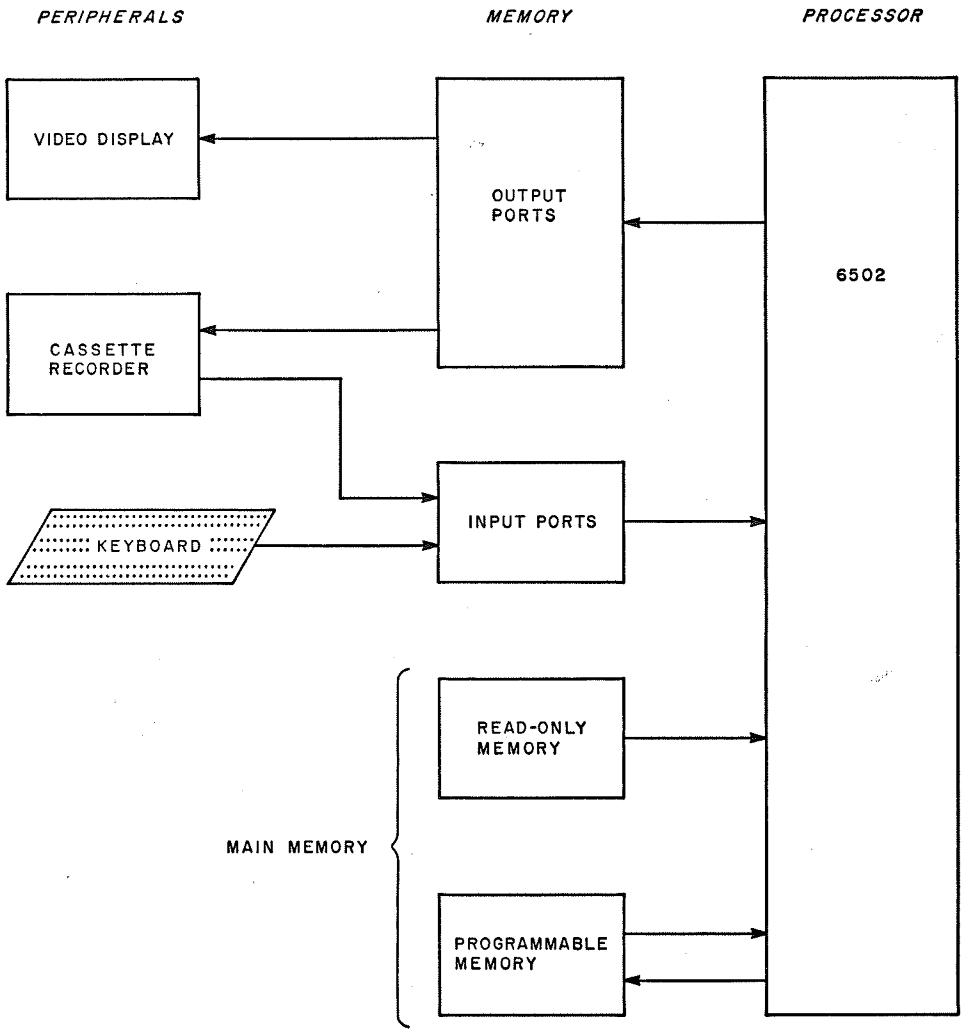

Figure 1.2: *A summary of the 6502 microprocessor's access to data in main memory and through I/O (input and output) ports. The arrows indicate the flow of data.*

A video screen connected to your computer looks like memory to the 6502, so the 6502 can read from and write to the screen. The keyboard is scanned by I/O (input/output) ports that are decoded to look like any other programmable memory address, so the 6502 can look at the keyboard just by looking at a particular place in memory. Thus, the 6502 can interact directly with memory only, but because all I/O devices are mapped to addresses in memory, the 6502 can interact with the user. See figure 1.3.

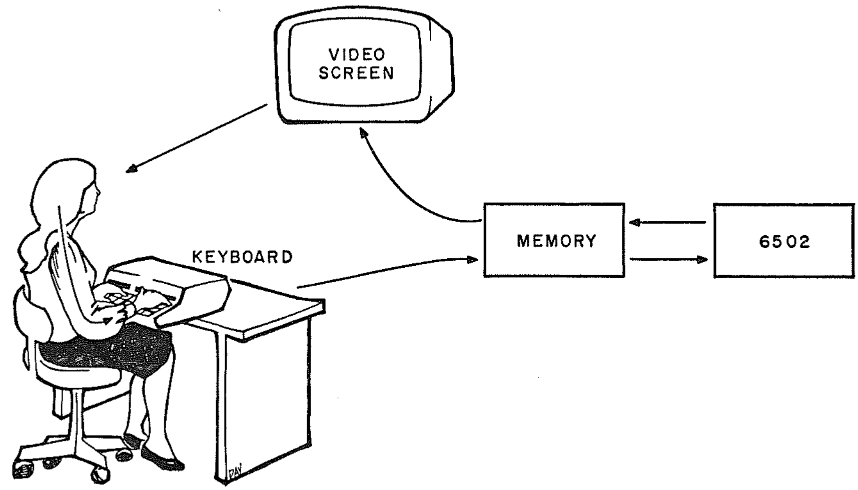

Figure 1.3: *How the 6502 interacts with the user. Arrows indicate the flow of data.*

The Operating System

Thus far we have discussed your machine's hardware. But the Apple, Atari OSI, and PET computers feature more than hardware. For example, all these computers have an operating system (stored in ROM) which includes the I/O software routines that are needed to use the screen and the keyboard. We are not particularly concerned with how these subroutines work, but I assume your system does have such routines.

There are many other subroutines in your computer's operating system. Your system's documentation should tell you what subroutines are available and provide their addresses. All of this means power for you, the programmer. The more you know about your computer, the more you can make it do. Because the software in this book was developed to run on a number of systems, I chose not to use routines available in your machine's ROM, no matter how powerful they might be, unless I could be sure that they would be available in the operating systems of the Apple, the Atari, the OSI, and the PET computers. In other words, the software in this book does not take full advantage of the power in your operating system. But the software you write, which need only run on your system, should exploit to the fullest the power of your computer's ROM routines.

BASIC

One of the most important features of your computer is the BASIC interpreter in ROM. This interpreter is a program that enables your computer to understand commands given in BASIC. Your system's documentation should tell you what commands are legal in the particular dialect of BASIC implemented on your machine. BASIC is an easy language to learn and you can do a lot with it.

Unfortunately, not every dialect of BASIC is the same. A program written in BASIC that runs on machine A may not run on machine B. BASIC is a common language, but not a standard one. Is there any language that *is* standard from system to system?

6502 Code

The central processor is the computer's heart. The Apple, Atari, OSI, and PET computers all use the 6502 microprocessor. Every microprocessor has a certain *instruction set*, or group of instructions, which the microprocessor can execute. These instructions are at a much lower level than the BASIC commands with which you may be familiar. For example, in BASIC you can have a single line in a program to PRINT "HELLO." It would take a sequence of many 6502 instructions to perform the same function.

However, a sequence of microprocessor instructions will run on any computer featuring that microprocessor. Thus, if you write a program consisting of 6502 instructions to perform some function, that program should run on any 6502-based computer. It won't run on an 8080-based computer, a Z80-based computer, or a 6800-based computer, but it should run on an Apple, a PET, an Atari, an OSI, or any other system built around a 6502. 6502 programs can also run much faster than equivalent programs written in BASIC and can be smaller than BASIC programs. The programs presented in this book are all written in 6502 code, and require only half of the memory available on a computer containing 8,000 bytes of programmable memory, thus leaving more than enough room for your own programs.

Chapter 2:

Introduction to Assembler

Ever watch a juggler or a good juggling team? The balls, pins, or whatever are in the air in such intricate patterns that you can hardly follow them, let alone duplicate the performance yourself. It's beautiful, but not magic; just an application of some simple rules. I've learned to juggle recently, and although I'm still a rank beginner, I've taught my two hands to keep three balls moving through the air. Yet neither hand knows very much. A hand will toss a ball into the air, and then it will catch a ball. The other hand will toss a ball into the air, and then it will catch a ball. That's all. My hands perform only two operations: toss and catch. Yet with those two primitive operations I can put on a pleasant little performance.

Assembly-language programming is not so different from juggling. Like juggling, programming enables you to put on an impressive or baffling performance. In its simplest terms, juggling is nothing more than taking something from one place and putting it someplace else. The same thing is true of the central processor: the 6502 takes something from one place and puts it someplace else.

In fact, programming the 6502 is easier than juggling in several ways. First, the 6502 is obviously much faster than even the most skillful juggler. In the time it takes me to pick up a ball with one hand and place that ball somewhere else, the 6502 can get something from one place and put it someplace else hundreds of thousands of times. Sleight of hand requires quickness, and the 6502 is quick.

The 6502 even gives me a helping hand. When I try to juggle, I must keep the balls moving with nothing but my two hands. But my home computer has three hands (registers A, X, and Y in the 6502) and thousands of pockets (8,000 bytes or more of programmable memory).

A byte is 8 bits of data that may be loaded together into a register. A register holds 1 byte. Each location in memory holds 1 byte. The 6502 can affect only 1 byte in one operation. But because the 6502 can perform hundreds of thousands of operations each second, it can affect hundreds of thousands of bytes each second.

Binary

In the final analysis, any value is stored within the computer as a series of bits. If we wish, we may specify a byte by its bit pattern: such a representation uses only ones and zeroes, and is called binary. For example, the number 25 in binary is 00011001.

In binary, each bit indicates the presence or absence of some value. Each bit represents twice as much value, or significance, as the bit to its right, so the right-most bit is the least significant, and the left-most bit is the most significant. Table 2.1 gives the significance of each bit in an 8-bit byte:

Table 2.1: *Bit significance in an 8-bit byte.*

Bit Number:	b7	b6	b5	b4	b3	b2	b1	b0
Bit Significance:	128	64	32	16	8	4	2	1

The right-most bit (called bit 0) tells us whether we have a one in our byte. The bit to its left (bit 1) tells us whether we have a two; the bit to *its* left tells us whether we have a four...and the leftmost bit (bit 7) tells us whether we have a 128 in our byte.

To determine the bit pattern for a given value — say, 25 — determine first what powers of two must be added to equal your value. For instance, $25 = 16 + 8 + 1$, so 25 in binary is 00011001.

Twenty-five can be expressed in other ways as well. Rather than specify every number as a pattern of eight ones and zeros, we often express numbers in hexadecimal representation.

Hexadecimal

Unlike binary, which requires a group of eight characters to represent an 8-bit value, hexadecimal notation allows us to represent an 8-bit value with a group of only two characters. These characters are not limited to 0 and 1, but may include any digit from 0 to 9, and any letter from "A" to "F." That gives us a set of sixteen characters, which is just right because we want to represent numbers in base 16.

(Hexadecimal stands for 16: hex for six, and decimal for ten. Six plus ten equals sixteen.)

To represent a byte in hexadecimal notation, divide the 8-bit byte into two 4-bit units (sometimes called *nybbles*). Each of these 4-bit units has a value of from 0 to 15 (decimal), which we express with a single hexadecimal digit. A decimal 10 is a hexadecimal $A. (The dollar sign indicates that a number is in hexadecimal representation.) Table 2.2 gives the conversions of decimal to hexadecimal for decimal numbers 0 thru 15.

Table 2.2: *Hexadecimal character set.*

Hexadecimal Character		Decimal Equivalent
$0	=	0
$1	=	1
$2	=	2
$3	=	3
$4	=	4
$5	=	5
$6	=	6
$7	=	7
$8	=	8
$9	=	9
$A	=	10
$B	=	11
$C	=	12
$D	=	13
$E	=	14
$F	=	15

Appendix A1, *Hexadecimal Conversion Table,* shows the hexadecimal representation of every number from 0 to 255 decimal.

In this book, object code, the only code that the machine can execute directly, will generally be presented in hexadecimal, and a thorough understanding of hexadecimal will help you to interpret instructions and follow some of the 6502's actions. Even the sketchiest understanding of hexadecimal math, however, should be sufficient for you to follow and use the programs in this book.

ASCII Characters

Instead of a number from 0 to 255, an 8-bit byte can be used to represent an upper or lower case letter of the alphabet, a punctuation mark, or a printer-control character such as a carriage return. A string of such bytes may represent a word, a message, or even a complete document. Appendix A2, *ASCII Character Codes*, gives the hexadecimal value for any ASCII character. ASCII stands for *A*merican *S*tandard *C*ode for *I*nformation *I*nterchange, and is the closest thing the industry has to a standard set of character codes. If you want to store the letter "A" in some location in memory, you can see from Appendix A2 that you must store a $41 in that location.

Whether a given byte is interpreted as a number, an ASCII character, or something else depends entirely on the program using that byte. Just as beauty is in the eye and mind of the beholder, so is the meaning of a given byte determined by the program that sees and uses it.

The Instruction Cycle

A microprocessor such as the 6502 can't do anything without being told. It only knows 151 instructions, called opcodes (operation codes). Each opcode is 1 byte long. An opcode may command the 6502 to take something from one register and to put it someplace in memory, to load some register with the contents of some location in memory, or to perform some other equally simple operation. See Appendix A4 for a list of opcodes for the 6502 microprocessor.

What do 6502s do all day? They work while programmers play. The 6502 gets an opcode, performs the specified operation, gets the next opcode, performs the specified operation, gets the next opcode, performs the...

You get the picture.

How does the 6502 know where to find the next opcode? The 6502 has a 16-bit register called the PC (program counter). The PC holds the address of some location in memory. When the 6502 starts its instruction cycle, it gets the opcode stored at the memory location specified by the PC. Then it performs the operation specified by that opcode. When it has executed that instruction, it makes the PC point to the next opcode and starts on a new instruction cycle by getting the opcode whose address is now in the PC.

Figure 2.1 shows a flowchart for the instruction cycle of the 6502 microprocessor.

"That's it? That's all the 6502 does?" you ask.

That's it. But with the right program in memory, we can make the 6502 dance.

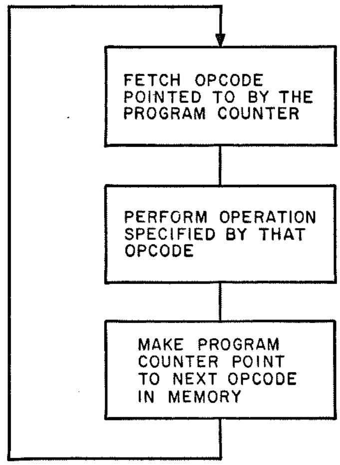

Figure 2.1: *The 6502 instruction cycle.*

Machine Language

A machine-language program is nothing more than a series of machine-language instructions stored in memory. If the PC in the 6502 can be made to hold the address of the start of your program, then we say that the PC is *pointing* to your program. When the 6502 starts its instruction cycle, it will *fetch* the first opcode in your program, and then perform the operation specified by that opcode. At this point, we say that your program is *running*.

Each machine-language instruction is stored in memory as a 1-byte opcode, which may be followed by 1 or 2 bytes of operand. Thus, a 6502 machine-language program might be "A9 05 20 02 04 A2 F5 60."

Just a bunch of numbers! (Hexadecimal numbers, in this case.) But it is exactly these numbers that the machine understands; hence the term, machine language.

Assemblers

Machine language is easy to read — if you're a machine. But programmers are people. So programming tools called assemblers have been developed, which take more readable assembly-language *source code* as input and produce *listings* and *object code* as output. The listing is the assembler's output intended for a human reader. The object code is a series of 6502 machine-language instructions intended to be stored in memory and executed by the 6502.

For each chapter in this book that presents a program, there is an appendix at the back of the book containing an assembler listing and a hexdump of the same program. The assembler listing includes both source and object code, making it easy for you to read the program; the hexdump shows you what the object code for that program actually looks like in your computer's memory. Figure 2.2 shows how an assembler is used to produce an assembler listing for the programmer and object code for the processor.

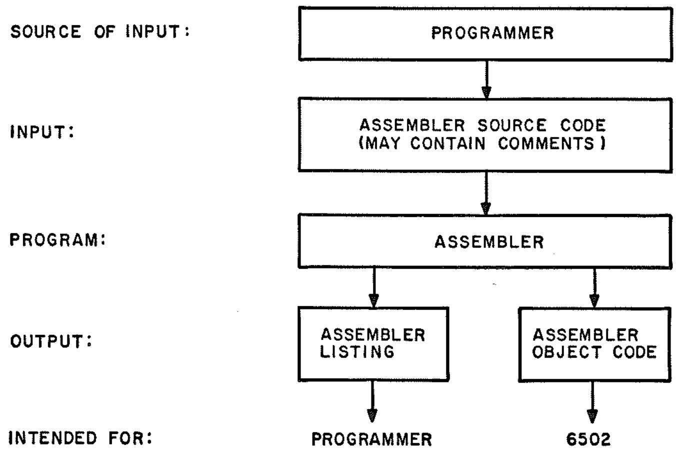

Figure 2.2: *From programmer to object code. The assembler takes source code as input and produces an assembler listing and object code as output.*

The programs in this book have all been produced on the OSI 6500 Assembler/Editor, running under the OSI 65-D Disk Operating System, on an OSI C-IP machine with 24 K bytes of programmable memory and one 5-inch floppy disk. It is likely that the source code presented in this book will assemble immediately or with only minor modification on other 6500 assemblers. (Incidentally, the source code in each chapter of this book should fit into the workspace of a computer with much less than 24 K bytes of user memory, if you delete many of the comments. But then, of course, your listings will be a lot less readable.)

But you don't write a listing; an assembler produces a listing. What you write is assembly-language source code.

Source Code

An assembly-language source program consists of one or more lines of assembly-language source code. A line of assembly-language source code consists of up to four fields:

LABEL MNEMONIC OPERAND COMMENT

The mnemonic, required in all cases, is a group of three letters chosen to suggest the function of a given machine-language instruction. For example, the mnemonic *LDA* stands for *L*oa*D* *A*ccumulator. *LDX* stands for *L*oa*D* *X* register. *TXA* means *T*ransfer the *X* register to the *A*ccumulator. 6502 mnemonics are not nearly as meaningful as BASIC commands, but they're a big improvement over the machine-language opcodes. See Appendix A3 for a list of 6502 mnemonics.

Some operations require an operand field. For example, the operation *load accumulator* requires an operand, because the line of source code must specify what you wish to load into the accumulator.

The label and comment fields are optional. A label lets you operate on some location in memory by a name that you have assigned to it. Comments are not included in the object code that will be assembled from your program, but they make your source code and your listings much more meaningful to a human reader. When you write a program, even if no one but yourself will ever read it, try to choose your labels and comments so that someone else can understand the purpose of each part of the program. Such careful documentation will save you a lot of time weeks or months down the road, when you might otherwise reread your program and have no idea why you included some unlabeled, uncommented line of source code.

Loading a Register

Let's write a simple program to load a register with a number — say, to load the accumulator with the number "10." Since we want to load the accumulator, we'll use the LDA instruction. (If we wanted to load the X register, we would use the LDX instruction, and if we wanted to load the Y register, we'd use LDY.) We know what mnemonic to write into our first line of source code. But a glance at Appendix A6, *6502 Opcodes by Mnemonic and Addressing Mode*, shows that LDA has many addressing modes. What operand shall we write into this line of source code?

We know that we want to load the accumulator with a "10," and not with any other number, so we can use the immediate addressing mode to load a "10" directly into the accumulator. We'll use a "#" sign to indicate the immediate mode:

Example 1

```
LDA #10
```

Example 1 is a legitimate line of source code containing only two fields: a mnemonic and an operand. The mnemonic, LDA, means "load the accumulator." But load it with what? The operand tells us what to load into the accumulator. The "#" sign specifies that this operation is to take place in the immediate mode, which means we want to load the accumulator with a constant to be found in this line of source code, rather than with data or a variable to be found in some location in memory. Then the operand specifies the constant to be loaded into the accumulator, in this case "10."

Constants

A constant is any value that is known by the programmer and "hard-wired" into the code. A constant does not change during the execution of a program. If a value changes during the execution of a program, then it is a variable, and one or more memory locations must be allocated to hold the current value of each variable.

There are several kinds of constants. Any number is a constant. The number "7," for example, is a constant: a seven now will still be a seven this afternoon. A character is another kind of constant: the letter "A" will still be the letter "A" tomorrow. But a variable, such as one called FUEL, will change during the course of a program (such as a lunar lander simulation), so it is not a constant.

In Example 1, note that the "#" sign is the only punctuation in the operand field. In the absence of special punctuation marks (such as the dollar sign indicating a hexadecimal number and the apostrophe indicating an ASCII character representation), any numbers given in this book are in decimal.

What object code will be assembled from this line of source code? Let's hand-assemble it and see. Appendix A6 shows us that the opcode for load accumulator, immediate mode, is $A9. So the first byte of object code for this instruction will be $A9. The second byte must specify what the 6502 should load into the accumulator. We want to load register A with a decimal 10, which is $0A. So the object code assembled from Example 1 is: A9 0A.

When these 2 bytes of object code are executed by the 6502, it will result in the accumulator holding a value of $0A, or decimal 10. In effect, we've just told a juggler: put a "10" in your right hand.

What if we wanted to load the accumulator with the letter "M," rather than with a number? We'd still use LDA to load the accumulator, and we'd still use the immediate mode of addressing, specifying in the operand the constant to be loaded into the accumulator. Either of the following two lines of source code will work:

Example 2

LDA #' M

or

LDA #$4D

In each line of source code above, the mnemonic and the "#" sign tell us we're loading the accumulator in the immediate mode — ie: with a constant. The operand following the "#" sign specifies the constant. An apostrophe indicates that an ASCII character follows, whereas a "$" sign indicates that a hexadecimal number follows. Appendix A2 shows that an ASCII "M" = $4D; they are simply two representations of the same bit pattern. So the two lines of source code above are equivalent; they will both assemble into the same object code: A9 4D.

Which of the two lines of source code is more readable? If a constant will be used in a program as an ASCII character, then represent it in your source code as an ASCII character.

Storing the Register

Now let's say we want to store the contents of the accumulator someplace in memory. Every location in memory has a unique address (just like houses do), ranging from $0000 to $FFFF. Suppose we decide to store the contents of the accumulator at memory location $020C. We could do it with the following line of source code:

Example 3

STA $020C

Example 3 will assemble into these 3 bytes of machine language: 8D 0C 02.

According to the Appendix A6, the 6502 opcode for "store accumulator, absolute mode" (STA) is $8D.

When the 6502 fetches the opcode "8D," it knows that it must store the contents of the accumulator at the address specified by the next 2 bytes. This is why it is called absolute mode. Absolute mode is used when specifying an exact memory location in an instruction.

In the example above, that address seems wrong. It looks like the machine-language operand is specifying address $0C02, because the bytes are in that order: "0C" followed by "02." But we want to operate an address $020C. Is something wrong here?

Low Byte First

You and I might think something is wrong when the address $020C is written as an "0C" followed by an "02" but you and I are people. We don't think like the 6502. When you and I write a number, we tend to write the most significant digit first and the least significant digit last. But the 6502 doesn't work that way. When the 6502 interprets two sequential bytes as an address, the first byte must contain the less significant part of the address (the "low byte"), and the second byte must contain the more significant part of the address (the "high byte"). All addressing modes that require a 2-byte operand require that the 2 bytes be in this order: less significant byte first, followed by the more significant byte.

However, not all addressing modes require a 2-byte operand.

Zero-Page Addressing

Memory is divided into pages, where a page is a block of 256 contiguous addresses. The page from $0000 to $00FF is called the zero page, because all addresses in this page have a high byte of zero. The zero-page addressing mode takes advantage of this fact. Source code assembled using the zero-page addressing mode requires only 1 byte in the operand, because the opcode specifies the zero page mode of addressing, and the high byte of the operand is unnecessary because it is understood to be zero. Thus, you can specify an address in the zero page by the absolute or by the zero-page addressing mode, but the zero-page mode will let you do it using one less byte.

If you want to use some location in the zero page to hold a number, you might decide to use location $00F4. We could write:

Example 4

```
STA $00F4
```

or

```
STA $F4
```

We could then assemble either line of source code using the absolute addressing mode: 8D F4 00. Or we could assemble either line of source code using the zero-page mode: 85 F4.

The opcode "85" means "store accumulator, zero page." Where in the zero page? At location $F4 in the zero page, the same location whose absolute address is $00F4.

Symbolic Expressions

Let's say you want to copy the 3 bytes at memory locations $0200, $0201, and $0202 to $0300, $0301, and $0302, respectively. We could write these lines of source code:

Example 5

```
LDA $0200
STA $0300
LDA $0201
STA $0301
LDA $0202
STA $0302
```

This alternately loads a byte into the accumulator, then stores the contents of the accumulator into another byte in memory. Note that loading a register from a location in memory changes the register, but leaves the contents of the memory location unchanged.

Or we could write the following code, which refers to addresses as symbolic expressions:

Example 6

```
1   ORIGIN = $0200
2   DEST   = $0300
3   LDA    ORIGIN
4   STA    DEST
5   LDA    ORIGIN + 1
6   STA    DEST + 1
7   LDA    ORIGIN + 2
8   STA    DEST + 2
```

In Example 6, lines 1 and 2 are assembler directives, which equate the labels "ORIGIN" and "DEST" with the addresses $0200 and $0300, respectively. Other lines of source code following these *equates* may then refer to these addresses by their labels, or refer to any address as a symbolic expression consisting of labels and, optionally, constants and arithmetic operators. The source code above will cause an assembler to generate exactly the same object code as the source code in Example 5, but Example 6, whose operands consist of symbolic expressions, is much more readable than Example 5, whose operands are given in hexadecimal.

Some Exercises

1) Write the 6502 instructions necessary to load the accumulator with the value 127, to load the X register with the letter "r," and to load the Y register with the contents of address $BO92.

2) Write the 6502 instructions necessary to copy the byte at address $0043 to the address $0092.

Chapter 3:

Loops and Subroutines

Indexed Addressing

Although readable, Example 6 is not very efficient, because it requires two lines of source code to move each byte. If we want to move 50 or 100 bytes must we then write 100 or 200 lines of source code?

Indexed addressing comes in quite handily here. Instead of specifying the absolute or zero-page address on which an operation is to be performed, we can specify a *base address* and an *index* register. The 6502 will add the value of the specified index registers to the base address, thereby determining the address on which the operation is to be performed. Thus, if we want to move 9 bytes from an origin to a destination, we could do it in the following manner, using the indexed addressing mode with X as the index register:

Example 7

```
        ORIGIN = $0200
        DEST   = $0300

INIT    LDX #0          Initialize X register to zero, so we'll start
                        with the first byte in the block.
GET     LDA ORIGIN,X    Get Xth byte in origin block.
PUT     STA DEST,X      Put it into the Xth position in the
                        destination block.
ADJUST  INX             Adjust X for next byte by incrementing
                        (adding 1) to the X register.
```

```
TEST      CPX #9           Done 9 bytes yet?
BRANCH    BNE GET          If not, go back and get next byte...
```

We will use Example 7 in the following sections to introduce several new instructions and addressing modes. Example 7 includes six lines of source code to move 9 contiguous bytes of data. If we tried to move 9 bytes of data with the techniques used in Examples 5 and 6, it would have taken eighteen lines of source code. So with indexed addressing, we've saved ourselves twelve lines of code. But how do these lines work? The lines are labeled so we can look at them one-by-one.

The instruction labeled INIT loads the X register in the immediate mode with the value zero. After executing the line INIT, the 6502 has a value of zero in the X register. We don't know anything about what's in the other registers.

GET loads the accumulator with the Xth byte above the address labeled ORIGIN. The first time the 6502 encounters this line, the X register will hold a value of zero, so the 6502 will load the accumulator with the zeroth byte above the address labeled ORIGIN (ie: it will load the accumulator with the contents of the memory location ORIGIN).

In any line of source code, a comma in the operand indicates that the operation to be performed shall use an indexed addressing mode. A comma followed by an "X" indicates that the X register will be the index register for an instruction, whereas a comma followed by a "Y" indicates that the Y register will be the index for an instruction. There are a number of indexed addressing modes. Two of these are absolute indexed and zero-page indexed. The line GET in Example 7 uses the absolute indexed addressing mode if ORIGIN is above the zero page; if ORIGIN is in the zero page then the line labeled GET can be assembled using the zero-page indexed addressing mode. Zero-page indexed addressing, like zero-page addressing, requires only 1 byte in the operand.

In zero-page indexed and in absolute indexed addressing, the operand field specifies a base address. The 6502 will operate on an address it determines by adding to the base address the value of the specified index register (X or Y). Only if the specified index register has a value of zero will the 6502 operate on the base address itself; in all other cases the 6502 will operate on some address higher in memory.

So we've loaded the accumulator with the byte at ORIGIN. Now the 6502 reaches the line labeled PUT in Example 7. This line tells the 6502 to store the accumulator in the Xth byte above DEST. We haven't done anything to change X since the line INIT set it to zero, so X still holds a value of zero. Therefore, the 6502 will store the contents of the accumulator in the zeroth byte above DEST (ie: in DEST itself).

At this point, we have succeeded in moving 1 byte from ORIGIN to DEST. X is still zero. Now comes the part that makes indexing worthwhile. The line labeled ADJUST is the shortest line of source code we've seen yet, consisting only of the mnemonic INX, which means "increment the X register." Since the X register was zero, when this line is executed the X register will be left holding a value of one.

Compare Register

In Example 7, the line labeled TEST compares the value in the X register with the number "9." There are three compare instructions for the 6502, one for each register. CMP compares a value with the contents of the accumulator; CPX compares a value with the contents of the X register, and CPY compares a value with the contents of the Y register.

We can use these compare instructions to compare any register with any value in memory, or, in the immediate mode, to compare any register with any constant. Such comparisons enable us to test for given conditions. For example, in Example 7, the line labeled TEST tests to see if we've moved 9 bytes yet. If the X register holds the value "9," then we have moved 9 bytes. (Walk through the loop yourself. When you have moved the zeroth through the eighth bytes above ORIGIN to the zeroth through the eighth positions above DEST, then you have moved 9 bytes.)

A compare instruction never changes the contents of a register or of any location in memory. Thus, the X register does not change when the line labeled TEST is executed by the 6502. What may change, however, are some of the 6502's status flags.

Status Flags

In addition to the 6502's general-purpose registers (A, X, and Y), the 6502 contains a special register P, the processor status register. Individual bits in the processor status register are set or cleared each time the 6502 performs certain operations. These bits, or hardware flags, are:

C	bit 0: Carry Flag
Z	bit 1: Zero Flag
I	bit 2: Interrupt Flag
D	bit 3: Decimal Flag
B	bit 4: Break Flag
	bit 5: Undefined
V	bit 6: Overflow Flag
N	bit 7: Negative Flag

In this book, we will not discuss the use of all the flags in the processor status register. In this quick course in assembly-language programming, and in the software subsequently presented in this book, the three flags we will deal with are C, the carry flag; Z, the zero flag; and N, the negative flag.

A compare operation (CMP, CPX, or CPY) does not change the value of registers A, X, or Y, but it does affect the carry, zero, and negative flags.

For example, if a register is compared with an equal value, the zero flag, Z, will be set; otherwise, Z will be cleared. If an instruction sets bit 7 of a register or an address, the negative flag of the status register will also be set; conversely, if an instruction clears bit 7 of a register or an address, the negative flag will be cleared. Similarly, mathematical and logical operations set or clear the carry flag, which acts as a ninth bit in all arithmetic and logical operations. Table 3.1 summarizes the effects of a compare instruction on the status flags.

Table 3.1: *Status flags affected by compare instructions. Note that if you wish to test the status of the carry flag after a compare, you must set it (using the instruction SEC) before the compare. When testing the N flag, think of the inputs as signed 8-bit values.*

	Carry Flag*	Negative Flag	Zero Flag
Compare a register with an *equal* value and you	set C,	clear N, and	set Z.
Compare a register with a *greater* value and you	clear C,	clear N, and	clear Z.
Compare a register with a *lesser* value and you	set C,	clear N, and	clear Z.

Conditional Branching

We can have a program take one action or another, depending on the state of a given flag. For example, two instructions, BEQ, (*B*ranch on result *EQ*ual) and BNE (*B*ranch on result *N*ot *E*qual) cause the 6502 to *branch*, or jump to a new instruction, based on the state of the zero flag. An instruction which causes the 6502 to branch based on the state of a flag is called a conditional branch instruction. Other conditional branch instructions are based on the state of other status flags and are given in table 3.2.

---

*If you wish to test the status of the carry flag after a compare, you must set it (using the instruction SEC) before the compare.

Table 3.2: *Conditional branch instructions.*

Flag	Instruction	Description	Opcode
C	BCC	Branch if carry clear.	90
C	BCS	Branch if carry set.	B0
N	BPL	Branch if result positive.	10
N	BMI	Branch if result negative.	30
Z	BEQ	Branch if result equal. (Zero Flag set).	F0
Z	BNE	Branch if result not equal. (Zero flag clear.)	D0
V	BVC	Branch if overflow flag clear.	50
V	BVS	Branch if overflow flag set.	70

The line labeled TEST in Example 7 compares the X register to the value "9;" this sets or clears the zero flag. The line labeled BRANCH then takes advantage of the state of the zero flag, by branching back to the line labeled GET if the result of that comparison was not equal. But if Y did equal "9," then the result of the comparison would have been equal, and the 6502 would *not* branch back to GET. Instead, the 6502 would execute the instruction following the line labeled BRANCH.

Loops

Example 7 shows a program loop. We cause the 6502 to perform a certain operation many times, by initializing and then incrementing a counter, and testing the counter each time through the loop to see if the job is done.

There's a lot of power in loops. What would we have to add or change in Example 7 so that it moves not 9, but 90 bytes from one place to another? Happily, we wouldn't have to add anything, and we'd only have to change the operand in the line labeled TEST. Instead of comparing the X register with 9, we'd compare it with 90. See Example 8.

Example 8

Move 90 bytes from origin to destination.

```
ORIGIN = $0200
DEST   = $0300
```

```
INIT      LDX #0           Initialize X register to zero, so we'll start
                           with the first byte in the block.
GET       LDA ORIGIN,X     Get Xth byte in origin block.
PUT       STA DEST,X       Put it into the Xth position in the
                           destination block.
ADJUST    INX              Adjust X for next byte.
TEST      CPX #90          Done 90 bytes yet?
BRANCH    BNE GET          If not, get next byte...
```

Writing loops lets us write code that is not only compact, but easily tailored to meet the demands of a particular application. We couldn't do that, however, without indexing and branching.

Loops can be tricky, though. What's wrong with this loop?

Example 9

```
          ORIGIN = $0200
          DEST   = $0300

INIT      LDX #0           Initialize X register to zero, so we'll start
                           with the first byte in the block.
GET       LDA ORIGIN,X     Get Xth byte in origin block.
PUT       STA DEST,X       Put it into the Xth position in the
                           destination block.
TEST      CPX #9           Done 9 bytes yet?
BRANCH    BNE GET          If not, get next byte...
```

Examine Example 9 very carefully. How does it differ from Example 7? It lacks the line labeled ADJUST, which increments the X register. What will happen when the 6502 executes the code in Example 9? It will initialize X to zero; it will get a byte from ORIGIN and move it to DEST. Then it will compare the contents of register X to 9. Register X won't equal 9, so it will branch back to GET, where it will do *exactly what it did the first time through the loop*, because X will still equal zero. Until the X register equals 9, the 6502 will branch back to GET. But nothing in this loop will ever change the value of X! So the 6502 will sit in this loop forever, getting a byte from ORIGIN and putting it in DEST and determining that the X register does not hold a 9...

Now look at Example 10. Will it cause the 6502 to loop, and if so, will the 6502 ever exit from the loop? Why, or why not?

Example 10

```
        ORIGIN = $0200
        DEST   = $0300

INIT    LDX #0          Initialize X register to zero, so we'll start
                        with the first byte in the block.
GET     LDA ORIGIN,X    Get Xth byte in origin block.
PUT     STA DEST,X      Put it into the Xth position in the
                        destination block.
ADJUST  INX             Adjust X for next byte.
TEST    CPX #9          Done 9 bytes yet?
BRANCH  BNE INIT        If not, get next byte...
```

Relative Addressing

All conditional branch instructions use the relative addressing mode, and they are the only instructions to use this addressing mode. Like the zero page and zero-page indexed addressing mode, the relative addressing mode requires only a 1-byte operand. This operand specifies the relative location of the opcode to which the 6502 will branch if the status register satisfies the condition required by the branch instruction. A relative location of 04 means the 6502 should branch to an opcode 4 bytes beyond the next opcode, if the given condition is satisfied. Otherwise, the 6502 will proceed to the next opcode.

Because the operand in a conditional branch instruction is only 1 byte, it is not possible for a conditional branch instruction to cause a branch more than 127 bytes forward or 128 bytes backward from the current value of the program counter. (A branch backward is indicated if the relative address specified is negative; forward if it's positive. A byte is negative if bit 7 is set. A byte is positive if bit 7 is clear. Thus, a value of 00 is considered positive.) However, an instruction called JMP allows the programmer to specify an unconditional branch to any location in memory. Therefore, if we have a short conditional branch followed by an unconditional jump, we may achieve in two instructions a conditional branch to any location in memory.

Unconditional Branch

Just as BASIC has its GOTO command, which causes an unconditional branch to a specified line in a BASIC program, the 6502 has its JMP instruction, which unconditionally branches to a specified address. A program may loop forever by JMP'ing back to its starting point.

Look at Example 11. Unless a line of code within the loop causes the 6502 to branch to a location outside of the loop, the 6502 will sit in this loop forever.

Example 11

Endless Loop:

```
START   xxxxxxxxxx      some
        xxxxxxxxxx      instructions
        xxxxxxxxxx
        JMP START
```

Indirect Addressing

A JMP instruction may be written in either the absolute addressing mode or the indirect addressing mode. Absolute addressing is used in Example 11. The operand is the address to which the 6502 should jump. But in the indirect mode (which is always signified by parentheses in the operand field) the operand specifies the address of a *pointer*. The 6502 will jump to the address specified by the pointer; it will not jump to the pointer itself.

The line of code "JMP (POINTR)" will cause the 6502 to jump to the address specified by the 2 bytes at POINTR and POINTR+1. Thus, if POINTR = $0600, and the 6502 executes the instruction "JMP (POINTR)" when memory location $0600 holds $00 and $0601 holds $20, then the 6502 will jump to address $2000. (Remember, addresses are always stored in memory with the low byte first.)

How Branching Works

Incidentally, all branches, whether relative, absolute, or indirect, work by operating on the contents of the PC (program counter). Before any branch instruction is executed, the PC holds the address of the current opcode. A branch instruction changes the PC, so that in the next instruction cycle the 6502 will fetch not the opcode following the current opcode, but the opcode at the location specified by the branch instruction. Then execution will continue normally from the new address.

Relocatability

Often I implement short unconditional branches as:

```
CLC
BCC    PLACE
```

rather than as:

```
JMP    PLACE
```

This is because the first method (relying as it does on relative rather than absolute addressing) will still work even if you relocate the code in which it is contained. Making your code relocatable will save you time and trouble when you try to move your programs around in memory and still want them to work.

To relocate code containing the second example, you'd have to change the operand field because the absolute address of PLACE will have changed. To relocate code containing the first example, you wouldn't have to change a thing.

Subroutines

Perhaps the two most powerful instructions available to the assembly-language programmer are the JSR (*J*ump to *S*ub*R*outine) and the RTS (*R*e*T*urn from *S*ubroutine). These instructions (equivalent to GOSUB and RETURN in BASIC) enable us to organize chunks of code as building blocks called subroutines.

Think of the subroutine as a job. Your computer can do more work for you if it knows how to do more jobs. Once you teach the 6502 how to do a given job, you won't have to tell it twice. Let's say you're writing a program in which the same operation must be performed at various times within a program. In every location within your program where the operation is required, you could include code to perform that operation. On the other hand, you could write code in one place to perform that operation, but write that code as a subroutine, and then *call* that subroutine whenever necessary from the main, or calling program. A call to a subroutine causes that routine to execute. When finished, it returns to the instruction following the call in the main program.

It only takes one line of code to call a subroutine. JSR SUB will call the subroutine located at the address labeled SUB. After the 6502 fetches and executes the JSR opcode, the next opcode it fetches will be at the address labeled SUB, in this example. So far it looks like an unconditional JMP. The 6502 will fetch and execute opcodes from the addresses following SUB, until it encounters an RTS instruction.

When the 6502 fetches an RTS instruction, it returns to its caller, jumping to the first opcode following the JSR instruction that called the subroutine. In effect, when a line of code calls a subroutine, the 6502 remembers where it is before it jumps to the new location. Then when it encounters an RTS instruction, it knows the address to which it should return because it remembers where it came from. It then continues to fetch opcodes from the point following the JSR instruction. Figure 3.1 illustrates this procedure. Note that the same subroutine may be called from many different points in the same program, and will always return to the opcode following the JSR instruction that called it.

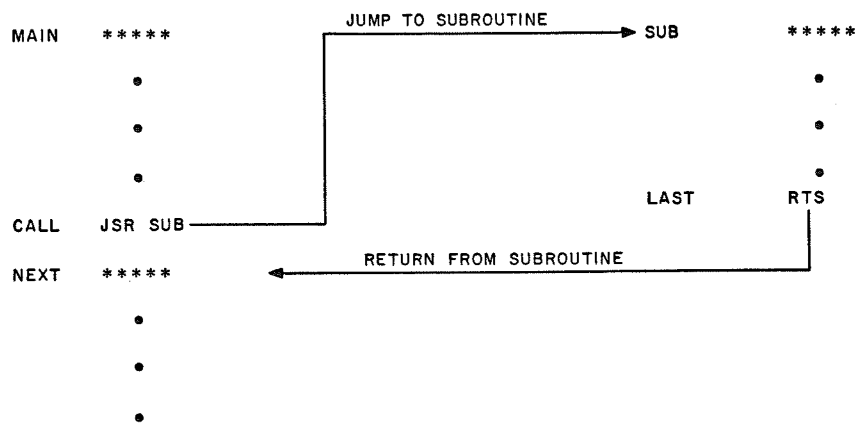

Figure 3.1: *Jump to and return from subroutine. When the processor encounters a JSR (jump to subroutine) instruction, the next instruction executed is the first instruction of the subroutine. Here, the subroutine SUB is called from MAIN. The last instruction executed in a subroutine must be an RTS (return from subroutine) instruction. Here, the instruction at label LAST in subroutine SUB returns control to the next instruction following the call to the subroutine in the main program, the instruction labeled NEXT. The subroutine SUB can be called anywhere in the program MAIN when the particular function of SUB is needed.*

Subroutines allow you to structure your software. With structured software, you can make changes to many programs just by changing one subroutine. If, for example, all programs that print characters do so by calling a single-character-print subroutine, then any time you improve that subroutine you improve the printing behavior of all your programs. Changing something only once is a tremendous advantage over having to change something in many different (usually undocumented) places within a piece of code. For these reasons, all of the software in this book uses subroutines.

Dummies

A *dummy subroutine* is a subroutine consisting of nothing but an RTS instruction. A line of code in a program can call a dummy subroutine and nothing will happen; the 6502 will return immediately, with its registers unchanged.

So why call a dummy subroutine?

A call to a dummy subroutine provides a "hook," which you may use later to call a functional subroutine. While developing a program, I may have many lines of code that call dummy subroutines. Later, when I write the lower-level subroutines, it's easy to change my program so that it calls the functional subroutines rather than the dummy subroutines. Trying to insert a subroutine call to a program lacking such a hook can make you wish for a "memory shoehorn," which might let you squeeze 3 extra bytes of code into the same address space.

The Stack

In addition to the addressing modes that enable the 6502 to access addressable memory, one addressing mode lets the 6502 access a 256-byte portion of memory called the *stack*.

You may think of this stack as a stack of trays in a cafeteria. The only way a tray can be added is to place it on top of the existing stack. Similarly, the only way to get a tray from the stack is to remove one from the top. This is the LIFO (Last-In, First-Out) method. The last tray placed onto the stack must be the first tray removed.

In our case, when an item is placed onto the top of the stack, it is called a *push*, and when an item is removed from the top of the stack, it is called a *pop*. The last item onto the stack is said to be at the *top* of the stack.

For example, let's say we want to place two items onto the stack. (Each item has an 8-bit value, perhaps a number or an ASCII character; see figure 3.2a.) First we push item 1 onto the stack, as illustrated in figure 3.2b. All positions above item 1 on the stack are said to be *empty*, the item 1 is on the top of the stack.

Now, push item 2 onto the stack (see figure 3.2c). What happens? Item 2 is now at the top of the stack, not item 1, although item 1 is still on the stack.

Next, to get item 2 back off the stack, we do a pop (see figure 3.2d). This makes item 1 the top of the stack again. Finally, another pop will remove item 1 from the stack, leaving the stack completely empty. Note that we had to pop item 2 from the stack before we could get to item 1 again. This is the LIFO principle.

The instruction PHA lets you push the contents of the accumulator onto the stack. PLA lets you load the accumulator from the top of the stack (a pop). PHP lets you push the processor status register onto the stack. PLP lets you load the processor status register from the stack.

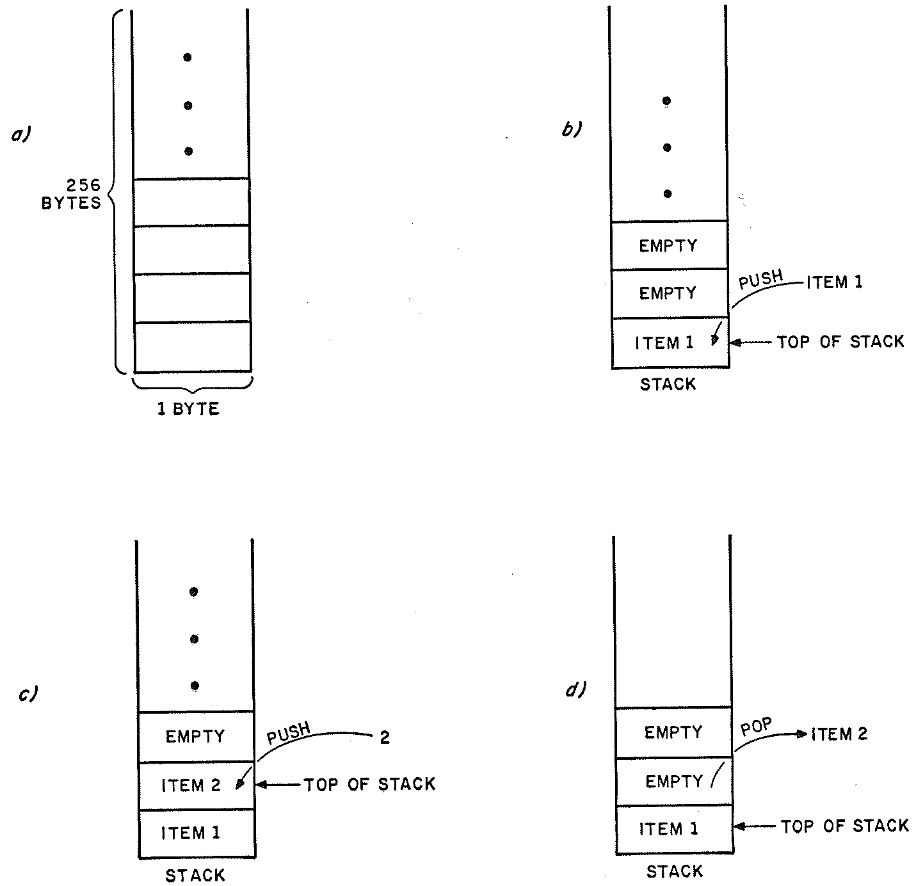

Figure 3.2: *Pushing and popping the stack.*

The stack is a very convenient "pocket" to use when you want to store one or a few bytes temporarily without using an absolute place in memory. Subroutines may pass information to the calling routines by using the stack, but be careful: if a subroutine pushes data onto the stack, and fails to pop that data from the stack before executing an RTS instruction, then that subroutine will *not* return to its caller. This happens because when the 6502 executes a JSR instruction, it pushes the return address—that is, the address of the opcode following the JSR instruction—onto the stack. A subroutine can return to its caller only because its return address is on the stack. If its return address is not at the top of the stack when the subroutine executes an RTS, it will not return to its caller. So a subroutine should always restore the stack before trying to return.

Chapter 4:

Arithmetic and Logic

Character Translation

As demonstrated by Examples 7 and 8, indexed addressing is handy for performing a given operation (such as a move) on a contiguous group of bytes. But it also has another important application: table lookup. For example, let's say you and a friend have decided to write notes to one another using a substitution code. For every letter, number, and punctuation mark in a message, you've agreed to substitute a different character. A "W" will be replaced with a "Y;" a semicolon may be replaced with a "9," etc.

You each have the same table showing you what to substitute for each character that may appear in a message. So you write a note to your friend in English, and then, using this table (which might be in the form of a Secret Agent Decoding Ring) you code, or encrypt, your note. You send the note in its encrypted form to your friend. Anyone else looking at the note would just see garbage, but your friend knows that a message can be found in it. So he gets his copy of the character translation table (which may be in *his* Secret Agent Decoding Ring), and he translates the encrypted message back into English, looking up the characters that correspond to each character in the coded message.

Children often enjoy coding and decoding messages in this way, but I find it about as much fun as filling out forms — which is no fun at all. Unfortunately, programming often involves character translation. Fortunately, I don't have to do it myself. I let my computer perform any necessary character translation by having it do what our two secret agents were doing: look up answers in a table.

Example 12
Character Translation Subroutine

```
XLATE   TAX            Use character to be translated as an in-
                       dex into the table.
        LDA TABLE,X    Look up value in table.
        RTS            Return to caller, bearing translated
                       character in A and original character in
                       X.
```

Transfer Register

In Example 12, the subroutine XLATE assumes when it is called that the accumulator holds the byte to be translated. This byte might be a letter, a number, a punctuation mark, a control code, or a graphic character, but however you think of it, it's an 8-bit value. Line 1 of XLATE transfers that 8-bit value from the accumulator to the X register, using the register-transfer instruction TAX.

Register-transfer instructions operate only on registers; they do not affect addressable memory. These instructions allow the contents of one register to be copied, or transferred, to another. The results of a transfer leave the source register unchanged, and the destination register holding the same value as the source register. The 6502's register-transfer instructions are:

TAX	Transfer accumulator to X register.
TAY	Transfer accumulator to Y register.
TXA	Transfer X register to accumulator.
TYA	Transfer Y register to accumulator.

Register transfers do not affect the status flags.

These instructions let you transfer A to X or Y, or to transfer X or Y to A. But how would you transfer X to Y, or Y to X? (Hint: it will take two lines of source code, each line an instruction from the list above.)

Table Lookup

In Example 12, line 2 of XLATE actually performs the character translation by looking up the desired data in a table. The label, TABLE, identifies the base address for a table that we've previously entered into memory. The indexed addressing mode allows line 2 to get the Xth byte above the base address (ie: to get the Xth byte of the table). When that line is executed, the table lookup is complete. The 6502 has looked up and now holds in the accumulator the Xth byte in the table. Now all the 6502 must do is return to its caller, bearing the translated character in A and the original character in X. It accomplishes this with the RTS instruction.

Now you can perform this character translation at any point in any program with just one line of source code:

```
JSR XLATE
```

Table lookup gives me great flexibility as a programmer. If a program uses a table lookup and for some reason I want the program to behave differently, I will probably only have to change some values in the table; it's unlikely that I'll have to change the table lookup code itself. If I've set up my table well, I might not have to change anything in the program except the data in the table.

Table lookup is therefore a very fast and flexible means of performing data translation. But the cost of that speed and flexibility can be size. You might be able to solve any problem with the right tables in memory, but not if you can't afford the memory necessary to hold all those tables. It's great when a program can just look up the answers it needs, but sometimes a program will actually have to *compute* its answers.

Arithmetic Operations

The 6502 can perform the following 8-bit arithmetical operations:

Shift
Rotate
Increment
Decrement
Add
Subtract

To understand how the 6502 operates on a byte, you must *think of the bits* in that byte. Even if the byte represents a number or a letter, don't think about what you can do to that number or letter. Think about what you can do to the pattern of bits in that byte.

What *can* you do to those bits?

Shift

You can shift the bits in a byte one position to the left or to the right. An ASL (Arithmetic Shift Left) operates on a byte in this manner: it moves each bit one bit to the left; it moves the leftmost bit (bit 7) into the carry flag, and it sets the rightmost bit (bit 0) to zero. See figure 4.1.

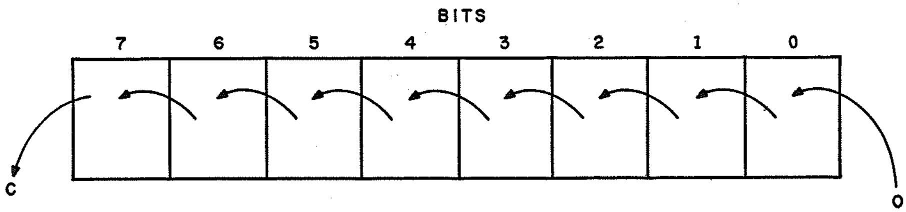

Figure 4.1: *Effect of the ASL instruction.*

For example, if the byte at location TMP has the following bit pattern:

address TMP 0 1 0 1 0 1 1 0

then after the instruction "ASL TMP" is executed, the data would look like:

address TMP 1 0 1 0 1 1 0 0

with the carry flag being set to the previous value of bit 7, in this case 0. If the same instruction is again executed, the data becomes:

address TMP 0 1 0 1 1 0 0 0

and the carry flag is set to 1.

A LSR (Logical Shift Right) has just the opposite effect of the ASL. All bits are shifted to the right towards the carry flag, introducing zeroes through bit 7. See figure 4.2.

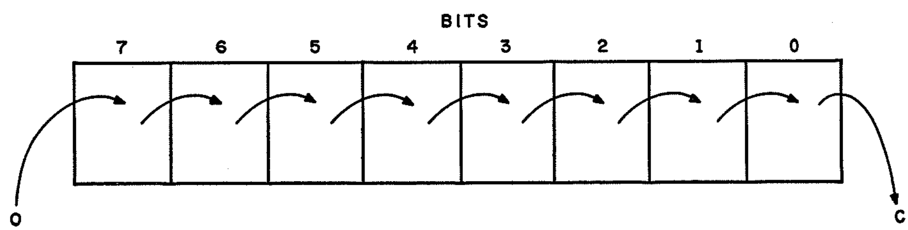

Figure 4.2: *Effect of the LSR instruction.*

For example, if the byte at location TMP is as originally given above, then after the instruction "LSR TMP" is executed, the data at TMP becomes:

address TMP 0 0 1 0 1 0 1 1

with the carry flag being set to the previous value of bit 0, in this case zero. If the same instruction is executed again, the data becomes:

address TMP 0 0 0 1 0 1 0 1

with the carry flag set to 1.

Because a number is represented in binary (each bit represents a successive power of two), some arithmetic operations are simple. To divide a byte by two, simply shift it right; to multiply a value in a byte by two, simply shift it left.

Rotate

You can also rotate the bits in a byte to the left or to the right *through* the carry flag. Unlike shifting, rotating a byte preserves all the information originally contained by a byte.

Figure 4.3 shows how a ROL (rotate left) instruction works. For instance, let's say the data at address TMP is originally the same as in previous examples:

address TMP 0 1 0 1 0 1 1 0

and let's say that the carry flag is set (ie: it holds a 1).

After a "ROL TMP" instruction is executed, the data becomes:

address TMP 1 0 1 0 1 1 0 1

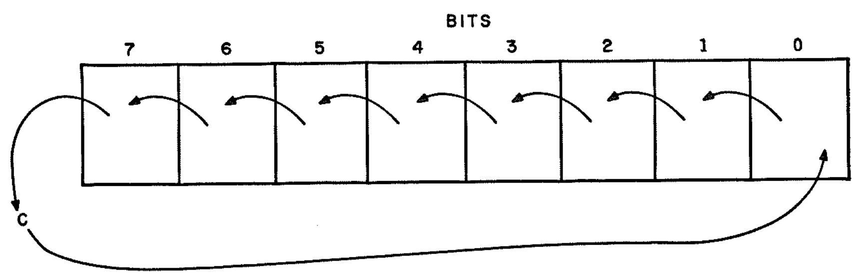

Figure 4.3: *Effect of the ROL instruction.*

and the carry bit is set to the previous value of bit 7, namely 1. Notice that bit 0 in TMP now holds the original contents of the carry flag, and the carry flag holds the original contents of bit 7. Otherwise, everything looks just the same as in the ASL operation. After a second execution of the instruction "ROL TMP," the data becomes:

address TMP 0 1 0 1 1 0 1 1

with the carry flag set to 1.

In a rotate left instruction, bit 0 is always set from the carry flag. (In the ASL instruction, bit 0 is always set to 0.) If this had been an ASL instruction, what would the bit pattern at TMP be?

Figure 4.4 shows how a ROR (rotate right) instruction works. It is similar to ROL, except that the carry flag is set *from* bit 0, and bit 7 is set from the carry flag.

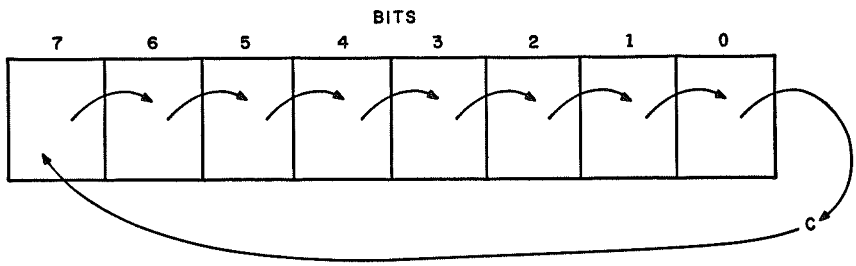

Figure 4.4: *Effect of the ROR instruction.*

Rotate a byte left nine times and you'll still have the original byte. The same is true if you rotate a byte right nine times. But *shift* a byte left nine times, or right nine times, and you know what you've got left? Nothing!

Increment, Decrement

You can increment or decrement a byte in three ways: using the INC and DEC instructions to operate on a byte in memory, using INX and DEX to operate on the X register, or using INY and DEY to operate on the Y register. None of these instructions affects the carry flag. They do affect the zero flag: Z is set if the result of an increment or decrement is zero; otherwise Z is cleared. The negative flag is set if the result of an increment or decrement is a byte with bit 7 set; otherwise N is cleared.

Note that if you increment a register or address holding $FF, it will hold zero. And similarly, if you decrement a register or address holding a zero, it will hold $FF.

You cannot increment or decrement the accumulator, but you can add or subtract a byte from the accumulator.

Addition

Example 13 shows how to add a byte from the location labeled NUMBER to the accumulator:

Example 13

CLC	Clear the carry flag.
ADC NUMBER	Add the contents of location NUMBER to the accumulator.

After these instructions are executed, the accumulator will hold the low 8 bits of the result of the addition. If, following the addition, the carry flag is set, then the result of the addition was greater than 255; if the carry flag is clear, then the result was less than 256, and, therefore, the accumulator is holding the full value of the result. Remember, the carry flag must be cleared before performing the ADC instruction.

Subtraction

Subtraction is as easy as addition. To subtract a byte from the accumulator, first set the carry flag (using the SEC instruction) and then subtract from the accumulator a constant or the contents of some address, using the instruction SBC (subtract with carry):

SEC	Set the carry flag.
SBC OPERND	Subtract from accumulator the value of OPERND.

If the operand is greater than the initial value of the accumulator, the subtract operation will clear the carry flag; otherwise the carry flag will remain set. In either case, the accumulator will bear the 8-bit result.

Thus, you clear the carry flag before adding and set the carry flag before subtracting. If the carry flag doesn't change state, then the accumulator bears the entire result. But if the addition or subtraction changes the state of the carry flag, then your result is greater then 255 (for an addition) or less than zero (for a subtraction).

Decimal Mode

The processor status register includes a bit called the *decimal flag*. If the decimal flag is set, then the 6502 will perform addition and subtraction in decimal mode. If the decimal flag is clear, then the 6502 will perform addition and subtraction in binary mode. Decimal mode means the bytes are treated as BCD (Binary Coded Decimal), meaning that the low 4 bits of a byte represent a value of 0 thru 9, and the high 4 bits of the byte represent a value of 0 thru 9. Neither *nybble* (4 bits) may contain a value of A-F. So, each nybble represents a decimal digit.

The instructions SED and CLD set the decimal flag and clear it, respectively. Unless you'll be operating with figures that represent dollars and cents, you won't need to use the decimal mode. All software in this book assumes that the decimal mode is not used.

Decimal 255 is the biggest value that can be represented by a binary-coded byte, but decimal 99 is the biggest value that can be represented by a byte using Binary Coded Decimal.

Logical Operations

What if you want to set, clear, or change the state of one or more bits in a byte without affecting the other bits in that byte? Input and output operations often demand such "bit-twiddling," which can be performed by the 6502's logical operations ORA, AND, and XOR.

Setting Bits

The ORA instruction lets you set one or more bits in the accumulator without affecting the state of the other bits. ORA logically OR's the accumulator with a specified byte, or *mask*, setting bit n in the accumulator if bit n in the accumulator is initially set *or* if bit n in the mask is set, or if both of these bits are set. A logical OR will leave bit n of the accumulator clear only if bit n is initially clear in both the accumulator and the mask. Table 4.1 shows a *truth table* for the logical operator OR. A truth table gives all possible combinations of 2 bits that can be operated upon (in this case, ORed) and the results of these combinations.

Table 4.1: *Truth table for the logical OR operand.*

Bit 1		Bit 2		Result
0	OR	0	=	0
0	OR	1	=	1
1	OR	0	=	1
1	OR	1	=	1

For example, suppose we executed the instruction "ORA #$80." Here the mask is $80, or the bit pattern 10000000. This instruction would therefore set bit 7 of the accumulator while leaving all other bits unchanged. So, if the accumulator had a value of 00010010 before the above instruction was executed, it would have the value of 10010010 afterwards.

Another example would be "ORA #3." Since a decimal 3 becomes 00000011 when converted to an 8-bit binary mask, the above instruction would set bits 0 and 1 in the accumulator, leaving bits 2 thru 7 unchanged.

How would you set the high 4 bits in the accumulator? The low 4 bits?

Clearing Bits

You can clear one or more bits in the accumulator without affecting the state of the other bits through the use of the AND instruction. AND performs a logical AND on the accumulator and the mask specified by the operand. AND will set bit n of the accumulator only if bit n of the accumulator is set initially *and* bit n is set in the mask. If bit n is initially clear in the accumulator or if bit n is clear in the mask, then AND will clear bit n in the accumulator. Table 4.2 gives the truth table for the logical AND operation.

Table 4.2: *The truth table for the logical AND.*

Bit 1		Bit 2		Result
0	AND	0	=	0
0	AND	1	=	0
1	AND	0	=	0
1	AND	1	=	1

For instance, the line of source code "AND #1" will clear all bits except bit 0 in the accumulator; bit 0 will remain unchanged. "AND #$F0" will clear the low 4 bits of the accumulator, leaving the high 4 bits unchanged. Select the right mask, and you can clear any bit or combination of bits in the accumulator without affecting the other bits in the accumulator.

Toggle Bits

The exclusive OR operation, XOR, lets you "flip," or toggle, one or more bits in the accumulator (ie: change the state of one or more bits without affecting the state of other bits). XOR will set bit n of the accumulator if bit n is set in the accumulator but not in the mask, or if bit n is set in the mask but not in the accumulator. If bit n has the same state in both the accumulator and in the mask, then XOR will clear bit n in the accumulator. Table 4.3 shows the truth table for this operation.

Table 4.3: *The truth table for the exclusive OR (XOR).*

Bit 1		Bit 2		Result
0	XOR	0	=	0
0	XOR	1	=	1
1	XOR	0	=	1
1	XOR	1	=	0

To toggle bit n in the accumulator, simply XOR the accumulator with a mask which has bit n set but all other bits clear. Bit n will change state in the accumulator, but all other bits in the accumulator will remain unchanged.

The logical operators, combined with the 6502's relative branch instructions, make it possible for a program to take one action or another depending on the state of a given bit in memory. Let's say you want a piece of code that will take one action (Action A) if a byte, called FLAG, has bit 6 set; yet take another action (Action B) if that bit is clear. The code of Example 14 shows one way to ignore all other bits in FLAG, and still preserve FLAG.

Example 14

```
LDA FLAG          Get flag byte.
AND #$40          Clear all bits but bit 6.
BEQ PLAN.B
```

```
PLAN.A          xxxxx          Take Action A, since bit 6 was set
                                in flag.
                  .
                  .
                  .
PLAN.B                         Take Action B, since bit 6 was
                                clear in flag.
```

What good are flags? Let me give an example. The flag on a rural mailbox may be either raised or lowered to indicate that mail is or is not awaiting pickup. Raising and lowering those flags requires a little bit of effort (no pun intended), but it enables the mail carrier to complete the route much more quickly than would be possible if every mailbox had to be checked every time around. Presumably, this provides better service for everyone on the route.

That mail carrier's routine is a very sophisticated piece of programming. If we think of the mail carrier as a person following a program, then we can see some of the power and flexibility that come from the use of flags.

The mail carrier's program has two parts: *What must be done at the post office* and *What must be done on the route*. At the post office, the mail carrier sorts the mail, bundles letters for the same address and puts the bundles for a given route into a mail sack in some order. This sorting at the post office means the mail carrier on the route can make his or her rounds more quickly, because no further sorting and searching is required. (We won't go into sorting and searching in this book; that's a volume in itself. For a helpful reference see Donald E Knuth's *Searching and Sorting*.)

Now comes the second part of the mail carrier's program: *What must be done on the route*. The mail carrier picks up the mail sack and leaves the post office. Driving down country roads, the mail carrier sees a mailbox ahead. *Do I have any mail for the people at this address? If so,* the mail carrier's mental program says, *I'll slow down and deliver it. But what if I don't have any mail now for these people? Do I just keep driving? Do I go to the next address?*

Not if I want to keep my job.

The mail carrier looks a little more closely at the mailbox. *Is the flag up or down? If it's down, I can just drive by, but if the flag is up I must stop and pick up the outgoing mail.*

A flag is just a single bit of information, but by interpreting and responding to the state of flags, even a simple program can respond to many changing conditions. If your computer has 8,000 bytes of programmable memory, that means it has 64,000 bits of memory. Conceivably, you could use most of those bits as flags, perhaps simulating the patterns of outgoing mail in a community of more than 50,000 households.

But you didn't buy a computer to play post office. And you know enough now to follow the programs presented in the following chapters. These programs will include examples of all the instructions and programming techniques presented in this very fast course in assembly-language programming. The programs in the following chapters will also give you some tools to use in developing your own programs.

(Incidentally, there is one 6502 instruction which doesn't do anything at all. The instruction NOP performs NO operation. Why would you want to perform no operation? Occasionally, it's handy to replace an unwanted instruction with a dummy instruction. When you want to disable some code, simply replace the unwanted code with NOP's. A NOP is represented in memory by $EA.)

Chapter 5:

Screen Utilities

Now let's consider how to display something on the video screen. On the Apple, Atari, OSI, and PET computers, the video-display circuitry scans a particular bank of memory, called the display memory. Every address in the display memory represents, or is mapped to, a different screen location (hence the term *memory-mapped display*). For each character in the display memory, the display circuitry puts a particular image, or graphic, on the screen (hence the term *character graphics*). To display a character in a given screen location, you need only store that character in the one address within display memory that corresponds to the desired screen location.

To know which address corresponds to a given screen location you must consult a display-memory map. Appendices B1 thru B4 describe how display memory is mapped on the Apple, Atari, OSI, and PET computers. Note that two different systems may have two different addresses for the same screen location. Also note how burdensome it can be to look up the addresses of even a few screen locations just to display a few characters on the video screen.

Rather than address the screen in an absolute manner, we'd like to be able to do so indirectly. Ideally, we'd like a software-controlled "hand" that we can move about the screen. Then we could pick up the character under the hand, or place a new character under the hand, without being concerned with the absolute address of the screen location under the hand at the moment. Such a hand can be implemented quite easily as a zero-page pointer.

Pointers

A pointer is just a pair of contiguous bytes in memory. Since 1 byte contains 8 bits, a pointer contains 16 bits, which means a pointer can specify any one of more than 65,000 (specifically: 2^{16}) different addresses.

A pointer can specify, or point to, only one address at a time. The low byte of a pointer contains the 8 LSB (least-significant bits) of the address it specifies, and the high byte of the pointer contains the 8 MSB (most-significant bits) of the address it specifies.

Let's say we want a pointer at location $1000. We must allocate 2 bytes for the pointer, which means it will occupy the bytes at $1000 and $1001. $1000 will hold the low byte, and $1001 will hold the high byte. If we want this pointer to specify address $ABCD, then we may set it as follows:

POINTR = $1000				This assembler directive equates the label POINTR with the value $1000. (It's POINTR and not POINTER only because the assembler used in preparing this book chokes on labels longer than six characters — a common, if arbitrary, limitation.)
LDA #$CD	A9	CD		Set the
STA POINTR	8D	00	10	low byte.
LDA #$AB	A9	AB		Set the
STA POINTR+1	8D	01	10	high byte.

Now POINTR points to $ABCD.

Although a pointer may be anywhere in memory, it becomes especially powerful when it's in the zero page (the address space from 0000 to $00FF). The 6502's indirect addressing modes allow a zero-page pointer to specify the address on which certain operations may be performed. A zero-page pointer must be located in the zero page, but it may point to any location in memory. For example, a zero-page pointer may be used to specify the address in which data will be loaded or stored. Since display memory looks like any other random-access memory to the processor, we may implement our television hand as a zero-page pointer.

TV.PTR

We want a zero-page pointer that can point to particular screen locations. Let's call it TV.POINTER, or TV.PTR for short. Whenever we examine or modify the screen, we'll do it through the TV.PTR.

Because the TV.PTR must be in the zero page, let's place it at $0000, meaning it will occupy the bytes at $0000 and at $0001. We can do that with the following assembler directive:

```
TV.PTR = $0
```

TV.PUT

The TV.PTR always specifies the current location on the screen. Thus, to display a graphic at the current location on the screen, we need only load the accumulator with the 8-bit code for that graphic and then execute the following two lines of code:

```
LDY #0            A0  00
STA (TV.PTR),Y    91  00
```

The two lines of above code are sufficient to display a given graphic in the current screen location. But what if you want to display a given *character* in the current screen location? The ASCII code for a character is not necessarily the same as your system's display code for that character's *graphic*. To display an "A" in the current screen location, we cannot simply load the accumulator with an ASCII "A" (which is $41) and then execute the two lines of above code, because the graphic "A" may have a different display code on your system. Instead of displaying an "A," we might display something else. Of the four computers considered in this book, only the Ohio Scientific Challenger I-P has a one-to-one correspondence between any character's ASCII code and that character's graphic code. The Atari, the PET, and the Apple computers lack such a one-to-one correspondence.

How then can we display a given ASCII character in the current screen location? We can do it by assuming that there exists a subroutine called FIXCHR, which will "fix" any given ASCII code, by translating it to its corresponding graphic or display code. FIXCHR will be different for each system, so we won't go into its details here (see the appendix pertaining to your computer for a description and listing of FIXCHR for your system). At this point we will assume only that FIXCHR exists, and that if we call it with an ASCII character in the accumulator, it will return with the corresponding display code in the accumulator.

We already know how to display a given graphic in the current screen location. With FIXCHR we now know how to display any given ASCII *character* in the current screen location. And since displaying any given ASCII character in the current screen location is something we're likely to do more than once, let's make it a subroutine. We'll call that subroutine TV.PUT since it will let us *put* a given ASCII character up on the TV screen:

```
TV.PUT    JSR FIXCHR           Convert ASCII character to your
                               system's display code for that character.
          LDY #0               Put that graphic in the
          STA (TV.PTR), Y      current screen location.
          RTS                  Return to caller.
```

The Screen Location

However, these examples of modifying and examining screen locations through the TV.PTR will work only if the TV.PTR is actually pointing at a screen location. Therefore, before executing code such as the examples given above, we must be sure the TV.PTR points to a screen location.

There are several ways to do this. If you want to write code that will run on only one machine (or on several machines whose display memory is mapped the same way), then you can use the immediate mode to set the TV.PTR to a given address on the screen. Let's say you want to set the TV.PTR to point to the third column of the fourth row (counting right and down from an origin in the upper-left corner). If you have an Ohio Scientific Challenger I-P, then you can consult your system's documentation and determine that address $D062 in display memory corresponds to your desired screen location. $D0 is the high byte of this screen location; $62 is the low byte of this screen location. Thus, you can set TV.PTR with the following lines of code:

```
LDA #$62        A9  62   Set
STA TV.PTR      85  00   low byte.
LDA #$D0        A9  D0   Set
STA TV.PTR+1    85  01   high byte.
      .
      .
      .
      .
      .
```

This code is fast and relocatable. But it's not very convenient to have to look up a display address every time we write code that displays something on the screen. It would be much more convenient if we could address the screen as a series of X and Y coordinates. Why not have a subroutine that sets the TV.PTR for us, provided we supply it with the desired X and Y coordinates?

TVTOXY

TVTOXY is a subroutine that sets the value of the TV.PTR to the display address whose X and Y coordinates are given by the X and Y registers. (Note that we count the columns and rows from zero.) To make the TV.PTR point to the third column from the left in the fifth row from the top, a calling program need only include the following code:

LDY #2	The leftmost column is column zero, so the third column is column two.
LDY #4	The topmost row is row zero, so the fifth row is row four.
JSR TVTOXY	Set TV.PTR to screen location whose X and Y coordinates are given by the X and Y registers.

.
.
.

How will TVTOXY work? We could have TVTOXY do just what we were doing: look up the desired address in a table. A computer can look up data in a table very quickly, but the speed may not be worth it if the table requires a lot of memory. If we don't mind waiting a little longer for TVTOXY to do its job, we can have TVTOXY *calculate* the desired value of TV.PTR, rather than look it up in a table. But how can you calculate the address of a given X and Y location on the screen?

You can't do it without data. But you don't need a large amount of data to determine the address of a given X,Y location in screen memory; you need only have access to the following facts:

HOME	The address of the character in the upper-left corner of the screen (ie: the lowest address in screen memory).
ROWINC	ROW INCrement: the address difference from one row to the next.

Knowing the values of HOME and ROWINC for a given system, you can calculate the address corresponding to any X,Y location:

HOME	Address of character in upper-left corner
+ X Register	+ X coordinate
+ (Y Register) × ROWINC	+ (Y coordinate) × ROWINC
TV.PTR	Address of screen location at column X, row Y.

Run through this calculation for several screen locations and compare the results with the addresses you look up in the display-memory map for your system. (Remember that we count columns and rows from zero, not from one.) Now if TVTOXY can run through this calculation for us, we'll never have to look at a display-memory map again; we can write all our display code in terms of cartesian coordinates.

But we shouldn't be satisfied with TVTOXY if it only runs through the above calculation. After all, what happens if TVTOXY is called and the Y register holds a very large number? If the Y register is greater than the number of rows on the screen, then the above calculation will set the TV.PTR to an address outside of display memory. We don't want that. Maybe a calling program will have a bug and call TVTOXY with an illegal value in X or in Y. If TVTOXY doesn't catch the error, the calling program may end up storing characters in memory that is not display memory. It might end up over-writing part of itself, which would almost certainly invite long and arduous debugging.

I hate debugging. I know I'm going to make mistakes, but I'd like my software to catch at least some bugs before they run amuck. So let's have TVTOXY check the legality of X and Y before blindly calculating the value of TV.PTR.

How can TVTOXY check the legality of X and Y? How big can X or Y get before it's too big? We need some more data:

TVCOLS	The number of columns on the display screen, counting from zero.
TVROWS	The number of rows on the display screen, counting from zero.

Now TVTOXY requires the following four facts about the host computer:

```
HOME
ROWINC
TVROWS
TVCOLS
```

If we store these facts about the host system in a particular block of memory, then TVTOXY need only consult that block of memory to learn all it needs to know about the screen. TVTOXY can then work as follows:

TVTOXY

```
TVTOXY    SEC               Is X out of range?
          CPX TVCOLS
          BCC X.OK          If not, leave it alone.
                            If X is out of range, give
          LDX TVCOLS        it its maximum legal value.
                            Now X is legal.

X.OK      SEC               Is Y out of range?
          CPY TVROWS
          BCC Y.OK          If not, leave it alone.
                            If Y is out of range, give
          LDY TVROWS        it its maximum legal value.
                            Now Y is legal.

Y.OK      LDA HOME          Set TV.PTR = HOME.
          STA TV.PTR
          LDA HOME+1
          STA TV.PTR+1

          TXA               Add X to TV.PTR.
          CLC
          ADC TV.PTR
          BCC COLSET
          INC TV.PTR+1
          CLC

COLSET    CPY #0            Add Y*ROWINC to TV.PTR.
          BEQ EXIT
LOOP      CLC
          ADC ROWINC
          BCC NEXT
```

```
            INC TV.PTR+1
NEXT        DEY
            BNE LOOP
EXIT        STA TV.PTR
            RTS                 Return to caller.
```

TVDOWN, TVSKIP, TVPLUS

Using TVTOXY, we can set TV.PTR to a screen location with any desired X,Y coordinates. But it would also be convenient to be able to modify TV.PTR relative to its current value. For example, after placing a character on the screen, we might want to make TV.PTR point to the next screen location to the right, or perhaps to the screen location directly below the current screen location. We might even want to make TV.PTR skip over several screen locations to make it point to "the nth screen location from here," where "here" is the current screen location. For these occasions, the subroutines TVDOWN, TVSKIP, and TVPLUS come in handy.

TVDOWN, TVSKIP, TVPLUS

```
TVDOWN      LDA ROWINC          Move TV.PTR down by one row.
            CLC
            BCC TVPLUS          Unconditionally branch.

TVSKIP      LDA #1              Skip one screen location by increment-
                                ing TV.PTR.

TVPLUS      CLC                 Add the contents of the accumulator
            ADC TV.PTR          to the two zero-page bytes
            BCC NEXT            comprising the TV.PTR.
            INC TV.PTR+1

NEXT        STA TV.PTR

            RTS                 Return to caller.
```

Note that the routines TVDOWN and TVSKIP make use of the routine TVPLUS, which assumes that the accumulator has been set to the number of locations to be skipped. For TVDOWN and TVSKIP, the accumulator is set to ROWINC and 1, respectively.

Right now TVPLUS might not seem long enough to be worth making into a subroutine. Any program that calls TVPLUS could perform the addition itself, at a cost of only a few bytes, and at a saving of several machine cycles in the process. However, we may make TVPLUS more sophisticated later on.

For example, we could enhance TVPLUS so it performs error checking automatically, to ensure that TV.PTR will never point to an address outside of screen memory. Such error checking would be very burdensome for every calling program to perform, but if and when we insert it into TVPLUS, every caller will automatically get the benefit of that modification.

VUCHAR

With TV.PUT we can display an ASCII character in the current screen location, and with TVSKIP we can advance to the next screen location. So why not combine the two, creating a subroutine that displays in the current screen location the graphic for a given ASCII character, and then automatically advances TV.PTR so it points to the next screen location? This would make it easy for a calling program to display a string of characters in successive screen positions. Since this subroutine will let the user *view* a *char*acter, let's call it VUCHAR:

```
VUCHAR   JSR TV.PUT      Display, in the current screen location,
                         the graphic for the character whose
                         ASCII code is in the accumulator.
         JSR TVSKIP      Advance to the next screen location.
         RTS
```

We could even squeeze VUCHAR into the code presented above for TVDOWN, TVSKIP, and TVPLUS, by inserting one new line of source code immediately above TVSKIP. (See Appendix C1, the assembler listing for the Screen Utilities, which also includes some error checking within TVPLUS.)

VUBYTE

With the screen utilities presented thus far, we can display a character on the screen in the current location, but we don't have a utility to display a *byte* in hexadecimal representation. Let's make one.

We'll call this utility VUBYTE, since it will let the user *view* a given *byte*. With VUBYTE, a calling program must take only three steps to display a byte in hexadecimal representation anywhere on the screen:

1) Set a zero-page pointer (TV.PTR) to point to the screen location where the byte should be displayed; 2) load the accumulator with the byte to be displayed; and then 3) call VUBYTE.

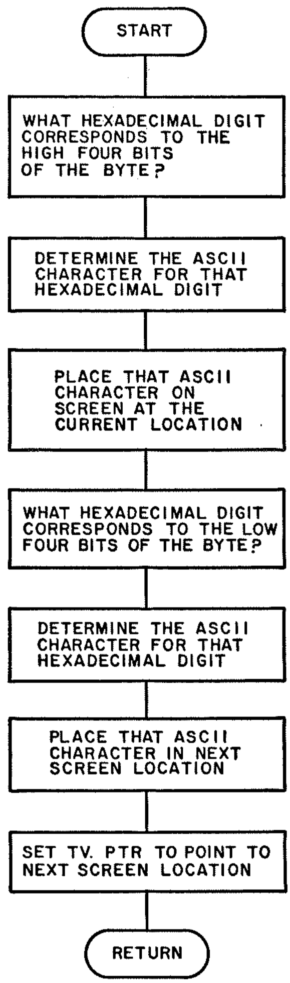

Figure 5.1: *Flowchart of the routine VUBYTE, which displays a byte in hexadecimal representation on the video screen.*

VUBYTE will display the given byte as two ASCII characters in the current position on the screen, and when VUBYTE returns, TV.PTR will be pointing to the screen location immediately following the two screen locations occupied by the displayed characters.

VUBYTE need only determine the ASCII character for the hexadecimal value of the 4 MSB (most-significant bits), store that ASCII character in the screen location pointed to by TV.PTR, then display the ASCII character for the hexadecimal value of the accumulator's 4 LSB (least-significant bits) in the next screen location. See figure 5.1 for a flowchart outlining this.

VUBYTE seems to be asking for a utility subroutine to return the ASCII character for a given 4-bit value. Let's call this subroutine ASCII. ASCII will return the ASCII character for the hexadecimal value represented by the 4 least-significant bits in the accumulator. It will ignore the 4 most-significant bits in the accumulator.

If we assume that ASCII exists, then we can write VUBYTE:

VUBYTE

VUBYTE	PHA	Save accumulator.
	LSR A LSR A LSR A LSR A	Move 4 MSB into positions occupied by 4 LSB.
	JSR ASCII	Determine ASCII for accumulator's 4 LSB (which *were* its 4 MSB).
	JSR VUCHAR	Display the ASCII character in the current screen location and advance to next screen location.
	PLA	Restore original value of accumulator.
	JSR ASCII	Determine ASCII for accumulator's 4 LSB (which were its 4 LSB).
	JSR VUCHAR	Display this ASCII character just to the right of the other ASCII character and advance to next screen location.
	RTS	Return to caller.

Of course, ASCII doesn't exist yet. So let's write it, and then VUBYTE should be complete.

ASCII

ASCII	AND #$0F	Clear the 4 MSB in accumulator.
	CMP #$0A BMI DECIML	Is accumulator greater than 9?
	ADC #6	If so, it must be A thru F. Add $36 to accumulator to convert it to corresponding ASCII character. (We'll add $36 by adding $6 and then adding $30.)
DECIML	ADC #$30	If accumulator is 0 thru 9, add $30 to it to convert it to corresponding ASCII character.
	RTS	Return to caller, bearing the ASCII character corresponding to the hexadecimal value initially in the 4 LSB of the accumulator.

TVHOME, CENTER

Now we can display a character or a byte at the current screen location, and we can set the current screen location to any given X,Y coordinates or modify it relative to its current value. It would also be handy if we could set the TV.PTR to certain fixed locations: locations that more than one calling program might need as points or origin. For example, a calling program might need to set the TV.PTR to the HOME location (position 0,0), or to the CENTER of the screen:

TVHOME, CENTER

TVHOME	LDX #0 LDY #0 JSR TVTOXY	Set TV.PTR to the leftmost column of the top row of the screen.
	RTS	Then return to caller.

```
CENTER    LDA TVROWS      Load A with total rows.
          LSR A           Divide it by two.
          TAY             Y now holds the number of the central
                          row on the screen.

          LDA TVCOLS      Load A with total columns.
          LSR A           Divide it by two.
          TAX             X now holds the number of the central
                          column on the screen.

                          Now X and Y registers hold X, Y coor-
                          dinates of center of screen.

          JSR TVTOXY      Set the TV.PTR to X,Y coordinates.

          RTS             Return to caller.
```

TVPUSH, TV.POP

The screen utilities presented thus far enable us to set or modify the current position on the screen. We might also want to save the current position on the screen and then restore that position later. We can do this by pushing TV.PTR onto the stack and then pulling it from the stack:

TVPUSH

```
TVPUSH    PLA             Pull return address from stack.

          TAX             Save it in X...
          PLA
          TAY             ...and in Y.

          LDA TV.PTR+1    Get TV.PTR
          PHA             and save
          LDA TV.PTR      it on
          PHA             the stack.

          TYA             Place return
          PHA             address back...
          TXA
          PHA             ... on stack.

          RTS             Then return to caller.
```

TVPOP

```
TV.POP      PLA             Pull return address from stack.
            TAX             Save it in X...
            PLA
            TAY             ...and in Y.

            PLA             Restore...
            STA TV.PTR      ...TV.PTR
            PLA               ...from
            STA TV.PTR+1        ...stack.

            TYA             Place return
            PHA             address back...
            TXA
            PHA             ... on stack.

            RTS             Then return to caller.
```

Now a calling program can save its current screen position with one line of source code: "JSR TVPUSH." That calling program can then modify TV.PTR and later restore it to its saved value with one line of source code: "JSR TV.POP."

CLEAR SCREEN

Now that we can set TV.PTR to any X,Y location on the screen, and display any byte or character in the current location, let's write some code to clear all or part of the screen. One subroutine, CLR.TV, will clear all of the video screen for us while preserving the zero page. A second routine, CLR.XY, will start from the current screen location and clear a rectangle, whose X,Y dimensions are given by the X,Y registers. Thus, a calling program can call CLR.TV to clear the whole screen; or a calling program can clear any rectangular portion of the screen, leaving the rest of the screen unchanged, just by making TV.PTR point to the upper left-hand corner of the rectangle to be cleared, and then calling CLR.XY with the X and Y registers holding, respectively, the width and height of the rectangle to be cleared.

```
CLR.TV    JSR TVPUSH        Save the zero-page bytes that will be
                            changed.
          JSR TVHOME        Set the screen location to upper-left cor-
                            ner of the screen.
```

```
         LDX TVCOLS        Load X,Y registers with
         LDY TVROWS        X,Y dimensions of the screen.
         JSR CLR.XY        Clear X columns, Y rows from current
                           screen location.
         JSR TV.POP        Restore zero-page bytes that were
                           changed.
         RTS               Return to caller, with screen clear and
                           with zero page preserved.

CLR.XY   STX COLS          Set the number of columns to be
                           cleared.
         TYA
         TAX               Now X holds the number of rows to be
                           cleared.
CLRROW   LDA BLANK         Load accumulator with your system's
                           graphic code for a blank.
         LDY COLS          Load Y with number of columns to be
                           cleared.
CLRPOS   STA (TV.PTR),Y    Clear a position by writing a blank into
                           it.
         DEY               Adjust index for next position in the
                           row.
         BPL CLRPOS        If not done with row, clear next posi-
                           tion...
         JSR TVDOWN        If done with row, move current screen
                           location down by one row.
         DEX               Done last row yet?
         BPL CLRROW        If not, clear next row...
         RTS               If so, return to caller.
COLS     .BYTE 0           Variable: holds number of columns to
                           be cleared.
```

There are many more screen utilities you could develop, but the utilities presented in this chapter are a good basic set. Now programs can call the following subroutines to perform the following functions:

ASCII:	Return ASCII character for 4 LSB in A.
CENTER:	Set current screen position to center of screen.
CLR.TV:	Clear the entire video display, preserving TV.PTR.
CLR.XY:	Clear a rectangle of the screen, with X,Y dimensions specified by the X,Y registers.
TVDOWN:	Move current screen position down by one row.

TVHOME:	Set current screen position to the upper-left corner of the screen.
TVPLUS:	Add A to TV.PTR.
TV.POP:	Restore previously saved screen position from stack.
TVPUSH:	Save current screen location on stack.
TV.PUT:	Display ASCII character in A at current screen location.
TVSKIP:	Advance to next screen location.
TVTOXY:	Set current screen position to X,Y coordinates given by X,Y registers.
VUBYTE:	Display A, in hexadecimal form, at current screen location. Advance current screen location past the displayed byte.
VUCHAR:	Display A as an ASCII character in current screen location; then advance to next screen location.

With these screen utilities, a calling program can drive the screen display without ever dealing directly with screen memory or even with the zero page. The calling program need not concern itself with anything other than the current position on the screen, which can be dealt with as a concept, rather than as a particular address hard-wired into the code.

Chapter 6:

The Visible Monitor

Hand Assembling Object Code

An assembler is a wonderful software tool, but what if you don't have one? Is it possible to write 6502 code without an assembler?

You bet!

Not only is it possible to write machine code by hand, but *all* of the software in this book was originally assembled and entered into the computer by hand. In fact, I hand assembled my code long after I had purchased a cassette-based assembler, because I could hand assemble a small subroutine faster than I could load in the entire assembler.

Hand assembling code imposes a certain discipline on the programmer. Because branch addresses must be calculated by counting forward or backward in hexadecimal, I tried to keep my subroutines very small. (How far can *you* count backward in hexadecimal?) I wrote programs as many nested subroutines, which I could assemble and test individually, rather than as monolithic, in-line code. This is a good policy even for programmers who have access to an assembler, but it is essential for any programmer who must hand assemble code.

Yet once you've written a program consisting of machine-language instructions, how can you enter it into memory? You can read your program on paper, but how can you present it to the 6502?

A program called a *machine-language monitor* allows you to examine and modify memory. It also allows you to execute a program stored in memory. The Apple and Ohio Scientific computers each feature a machine-language monitor in ROM (read-only memory). The Atari computers feature a machine-language monitor in a plug-in program cartridge. Your system's documentation should tell you how to use the features of your monitor, but let's take a closer look at one monitor in particular, the Ohio Scientific 65V monitor. Because it is stored in read-only memory in the OSI Challenger I-P, I will refer to it as the OSI ROM monitor.

A Minimal Machine-Language Monitor

You can invoke the OSI ROM monitor quite easily by pressing the BREAK key and then the "M" key. The monitor clears the video screen and presents the display shown in figure 6.1.

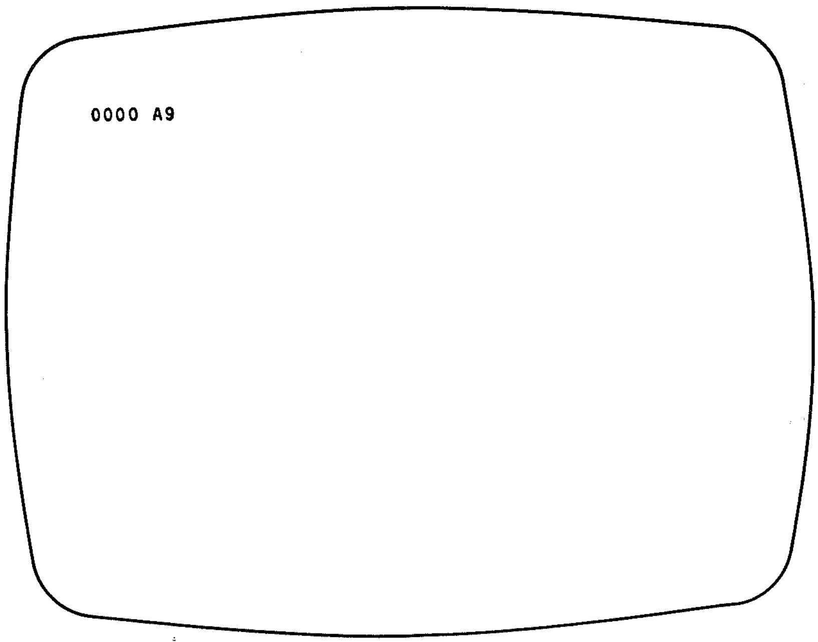

Figure 6.1: *Ohio Scientific ROM (read-only memory) monitor display.*

The display consists of two fields of hexadecimal characters: an address field and a data field. Figure 6.1 indicates that $A9 is the current value of address $0000.

The OSI ROM monitor has two modes: *address mode* and *data mode*. When the monitor is in address mode, you can display the contents of any address simply by typing the address on the keyboard. Each new hexadecimal character will roll into the address field from the right. To display address $FE0D, you simply type the keys F, E, 0, and then D.

To change the contents of an address, you must enter the *data* mode. When the

OSI ROM monitor is in the data mode, hexadecimal characters from the keyboard will roll into the data field on the screen. For your convenience, when the monitor is in the data mode you can step forward through memory (ie: increment the displayed address) by depresssing the RETURN key. Unfortunately, this convenience is not available in address mode, and neither mode allows you to step backward through memory (ie: to decrement the address field).

Beware: the OSI ROM monitor can mislead you. If the monitor is in the data mode and you type a hexadecimal character on the keyboard, that character will roll into the data field on the screen. Presumably that hexadecimal character also rolls into the memory location displayed on the screen. Yet, this might not be the case. In fact, the OSI ROM monitor displays the data you *intended* to store in an address, rather than the actual contents of that address. If you try to store data in a read-only memory address, for example, the OSI ROM monitor will confirm that you've stored the intended data in the displayed address, yet if you actually inspect that address (by entering address mode and typing in the address), you'll see that you changed nothing. This makes sense — you can't write to read-only memory. But the OSI ROM monitor leads you to think that you can.

The OSI ROM monitor can be confusing in other ways. For example, the display does not tell you whether you're in data mode or address mode; you've got to remember at all times which mode you last told the monitor to use. Furthermore, to escape from address mode you must use one key, while to escape from data mode you must use another key. Therefore you must always remember two escape codes as well as the current mode of the monitor.

Furthermore, the OSI ROM monitor does not make it very easy for you to enter ASCII data into memory. To enter an ASCII message into memory, you must consult an ASCII table (such as Appendix A2 in this book), look up the hexadecimal representation of each character in your message, and then enter each of those ASCII characters via two hexadecimal keystrokes. Then, once you've got an ASCII message in memory, the OSI ROM monitor won't let you read it as English text; you'll have to view that message as a series of bytes in hexadecimal format, and then look up, again in Appendix A2 or its equivalent, the ASCII characters defined by those bytes. That won't encourage you to include a lot of messages in your software — even though meaningful prompts and error messages can make your software much easier to maintain and use.

Finally, it is worth examining the way the OSI ROM monitor executes programs in memory. When you type "G" on the Ohio Scientific Challenger I-P, the OSI ROM monitor executes a JMP (unconditional jump) to the displayed address. That transfers control to the code selected, but it does so in such a way that the code must end with another unconditional jump if control is to return to the OSI ROM monitor. This forces you to write programs that end with a JMP, rather than subroutines that end with an RTS.

Programs that end with a JMP are not used easily as building blocks for other programs, whereas *subroutines* are incorporated quite easily into software structures of ever-greater power. So wouldn't it be nice if a machine-language monitor executed a JSR to the displayed address? This would call the displayed address as a subroutine, encouraging users to write software as subroutines, rather than as code that jumps from place to place. Such a monitor might actually encourage good programming habits, inviting the user to program in a structured manner, rather than daring the user to do so. In this chapter we'll develop such a monitor.

Objectives

If you've spent any time using a minimal machine-language monitor, you've probably thought of some ways to improve it. Based on my own experience, I knew that I wanted a monitor to be:

1) Accurate

The data field should display the actual contents of the displayed address, not the *intended* contents of that address.

2) Convenient

It should be possible to step forward or backward through memory, in any mode. It should also be possible to enter ASCII characters into memory directly from the keyboard, without having to look up their hexadecimal representations first, and it should be possible to display such characters as ASCII characters, rather than as bytes presented as pairs of hexadecimal digits.

3) Encourage Structured Programming

The monitor should *call* the displayed address as a subroutine, rather than *jump* to the displayed address. This will encourage the user to write subroutines, rather than monolithic programs that jump from place to place.

4) Simplify Debugging

The monitor should load the 6502 registers with user-defined data before calling the displayed address. Thus a user can initially test a subroutine with different values in the registers. Then, when the called subroutine returns, the monitor should display the new contents of the 6502 registers. Thus, by seeing how it changes or preserves the values of the 6502 registers, the user could judge the performance of the subroutine.

Because my objective was to make the 6502 registers visible to the user by displaying the 6502 registers before and after any subroutine call, I've chosen to call this monitor the *Visible Monitor*. Figure 6.2 shows its display format.

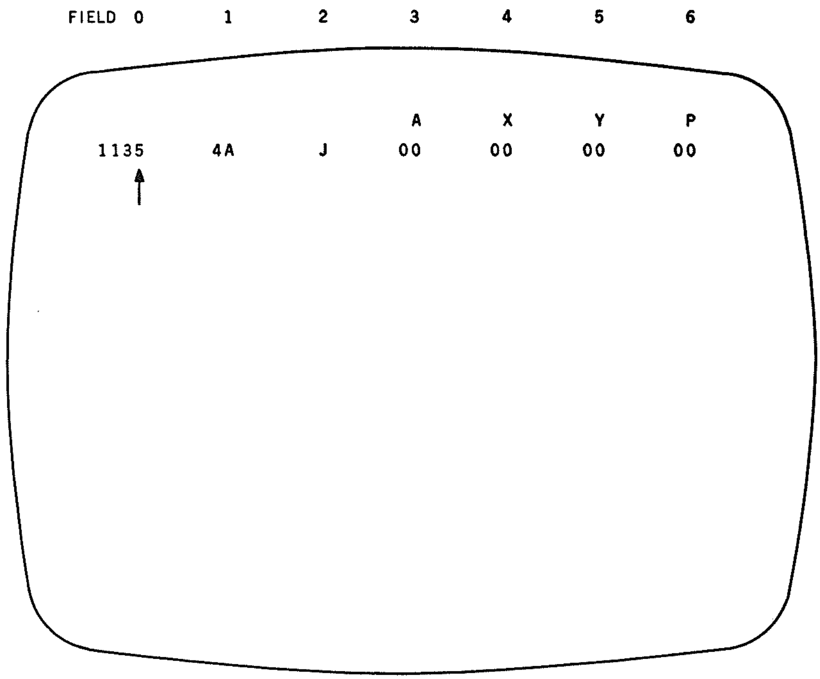

Figure 6.2: *Visible Monitor Display with fields numbered.*

VISIBLE MONITOR DISPLAY

The Visible Monitor Display

Notice that the display in figure 6.2 has seven fields, not two as in the OSI ROM monitor display. The first two fields (fields 0 and 1) are the same as the two fields in the OSI ROM monitor — that is, they display an address and a hexadecimal representation of the contents of that address. Field 2 is a graphic representation of the contents of the displayed address. If that address holds an ASCII character, then the graphic will be the letter, number, or punctuation mark specified by the byte. Otherwise, that graphic will probably be a special graphic character from your computer's nonstandard (ie: nonASCII) character set.

Fields 3 thru 6 represent four of the 6502 registers: A (the Accumulator), X (the X Register), Y (the Y Register), and P (the Processor Status Register). When you type

G to execute a program, the 6502 registers will be loaded with the displayed values before the program is called; when control returns to the monitor, the contents of the 6502 registers at that time will be displayed on the screen.

In addition to the seven fields mentioned above, the Visible Monitor's display includes an arrow pointing up at one of the fields. In order to modify a field, you must make the arrow point to that field. To move the arrow from one field to another, I've chosen to use the GREATER THAN (>) and LESS THAN (<) keys. Touching the GREATER THAN key will move the arrow one field to the right, and depressing the LESS THAN key will move the arrow one field to the left. (If my computer had a cursor pad, I would use the cursor-left and the cursor-right keys to move the arrow from field to field, but it doesn't have a cursor pad, so GREATER THAN and LESS THAN have to fill the bill. You may assign the field-movement functions to any keys on your system, but GREATER THAN and LESS THAN are reasonable choices, because they look like arrows pointing right and left, respectively.)

I've chosen to use the space bar to step forward through memory and the return key to step backward through memory, but you may choose other keys if you prefer (eg: the "+" and "−" keys). The space bar seems reasonable to me for stepping forward through memory, because on a typewriter I press the space bar to bring the *next* character into view; RETURN seems reasonable for stepping backward through memory because RETURN is almost synonymous with "back up," and that's what I want it for: to back up through memory. With such a display and key functions, we ought to have a very handy monitor.

Data

Before we develop the structure and code of the Visible Monitor, let's decide what variables and pointers it must have.

The Visible Monitor must have some way of knowing what address to display in field 0. It can do this by maintaining a pointer to the currently selected address. Because it will specify the currently selected address, let's call this pointer SELECT. Then, when the user presses the spacebar, the Visible Monitor need only increment the SELECT pointer. When the user presses RETURN, the Visible Monitor need only decrement the SELECT pointer. That will enable the user to step forward and backward through memory.

The user will also want to modify the 6502 register images. Since there are four register images shown in figure 6.2, let's have 4 bytes, one for each register image. If we keep them in contiguous memory, we can refer to the block of register images as REGISTERS, or simply as REGS (since REGISTERS is longer than six characters, the maximum label length acceptable to the assembler used in the preparation of this book).

Finally, the Visible Monitor must keep track of the current field. Since there can only be one current field at a time, we can have a variable called FIELD, whose value tells us the number of the current field. Then, when the user wants to select the next field, the Visible Monitor need only increment FIELD, and when the user wants to move the arrow to the previous field, the Visible Monitor need only decrement FIELD. If FIELD gets out of bounds (any value that is not 0 thru 6), then the Visible Monitor should assign an appropriate value to FIELD. The following code declares these variables in the form acceptable to an OSI 6500 Assembler:

Variables

```
SELECT      .WORD 0        This points to the currently selected
                           byte.
REG.A       .BYTE 0        REG.A holds the image of Register A
                           (the Accumulator).
REG.X       .BYTE 0        REG.X holds the image of Register X.
REG.Y       .BYTE 0        REG.Y holds the image of Register Y.
REG.P       .BYTE 0        REG.P holds the image of the Processor
                           Status Register.
FIELD       .BYTE 0        FIELD holds the number of the current
                           field.
            REGS = REG.A
```

Structure

I want to keep the Visible Monitor highly modular, so it can be easily extended and modified. I have therefore chosen to develop the Visible Monitor according to the structure shown in figure 6.3. Clearly, the Visible Monitor loops. It places the monitor *display* on the screen. It then *updates* the information in that display by getting a keystroke from the user and performing an action based on that keystroke. It does this over and over.

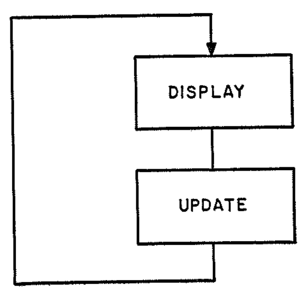

Figure 6.3: *A simple structure for interactive display programs.*

With this flowchart as a guide, we can now write the source code for the top level of the Visible Monitor:

VISMON

```
VISMON      PHP           Save caller's status flags.
LOOP        JSR DSPLAY    Put monitor display on screen.
            JSR UPDATE    Get user request and handle it.
            CLC
            BCC LOOP      Loop back to display...
```

This is only the top level of the Visible Monitor; it won't work without two subroutines: DSPLAY and UPDATE. So it looks as if we've traded the task of writing one subroutine for the task of writing two. But by structuring the monitor in this way, we make the monitor much easier to develop, document, and debug.

Which subroutine should we write first? Let's start with the DSPLAY module, since the display is visible to the user, and the Visible Monitor must meet the user's needs. Once we know how to drive the display, we can write the UPDATE routine.

Monitor Display

Figure 6.2 shows the display we want to present on the video screen. As you can see, this display consists of three lines of characters: the label line, the data line, and the arrow line. The label line labels four of the fields in the data line, using the characters A, X, Y, and P. The data line displays an address, the contents of that address (both in hexadecimal representation and in the form of a graphic), and then displays the values of the four registers in the 6502. Underneath the data line, the arrow line provides one arrow pointing up at one of the fields in the data line.

Since the display is defined totally in terms of the label line, the data line, and the arrow line, we are ready now to diagram the top level of monitor display. See figure 6.4.

With the flowchart in figure 6.4 as a guide, we can now write source code for the top level of the DSPLAY subroutine:

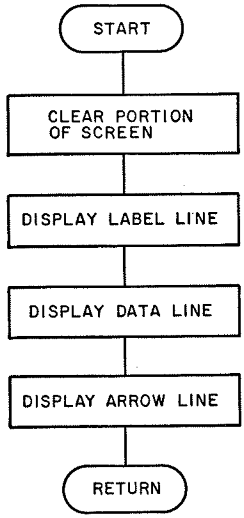

Figure 6.4: *Routine to display the monitor information.*

DSPLAY

```
DSPLAY      JSR CLRMON    Clear monitor's portion of screen.
            JSR LINE.1    Display the Label Line.
            JSR LINE.2    Display the Data Line.
            JSR LINE.3    Display the Arrow Line.
            RTS           Return to caller.
```

Now instead of one subroutine (DSPLAY), it looks as if we must write four subroutines: CLRMON, LINE.1, LINE.2, and LINE.3. But as the subroutines grow in number, they shrink in difficulty.

Before we put up any of the monitor's display, let's clear that portion of the screen used by the monitor's display. Then we can be sure we won't have any garbage cluttering up the monitor display.

Since we already have a utility to clear X columns and Y rows from the current location on the screen, CLRMON can just set TV.PTR to the upper-left corner of the screen, load X and Y with appropriate values, and then call CLR.XY. Here's source code:

```
CLRMON      LDX #2           Set TV.PTR to column 2, row 2 of
            LDY #2           screen.
            JSR TVTOXY
            LDX #25          We'll clear 25 columns
            LDY #3           and 3 rows.
            JSR CLR.XY       Here we clear them.
            RTS              Return to caller.
```

Display Label Line

The subroutine LINE.1 must put the label line onto the screen. We'll store the character string "A X Y P" somewhere in memory, at a location we may refer to as LABELS. Then LINE.1 need only copy 10 bytes from LABELS to the appropriate location on the screen. That will display the LABEL line for us:

LINE.1

```
LINE.1      LDX #13          X-coordinate of Label "A".
            LDY #2           Y-coordinate of Label "A".
            JSR TVTOXY       Place TV.PTR at coordinates given by
                             X,Y registers.
            LDY #0           Put labels on the screen:
            STY LBLCOL       Initialize label column counter.
LBLOOP      LDA LABELS,Y     Get a character and
            JSR VUCHAR       put its graphic on the screen.
            INC LBLCOL       Prepare for next character.
            LDY LBLCOL       Use label column as an index.
            CPY #10          Done last character?
            BNE LBLOOP       If not, do next one.
            RTS              Return to caller.
LABELS      .BYTE 'A X '     These are the characters
            .BYTE 'Y P'      to be copied to the screen.
LBLCOL      .BYTE 0          This is a counter.
```

Display Data Line

Displaying the data line will be more difficult than displaying the label line, for two reasons. First, the data to be displayed will change from time to time, whereas the labels in the label line need never change. Second, most fields in the data line display data in hexadecimal representation. To display 1 byte as two hexadecimal digits requires more work than is needed to display 1 byte as one ASCII character. However, we have a screen utility (VUBYTE) to do that work for us. In fact, we have enough screen utilities to make even the display of seven fields of data quite straightforward. Following, then, is the display data-line routine:

LINE.2

```
LINE.2     LDX #2           Load X register with X-coordinate for
                            start of data line.
           LDY #3           Load Y register with Y-coordinate for
                            data line.
           JSR TVTOXY       Set TV.PTR to point to the start of the
                            data line.
           LDA SELECT+1     Display high byte of the
           JSR VUBYTE       currently selected address.
           LDA SELECT       Display low byte of the
           JSR VUBYTE       currently selected address.
           JSR TVSKIP       Skip one space after address field.
           JSR GET.SL       Look up value of the currently selected
                            byte.
           PHA              Save it.
           JSR VUBYTE       Display it, in hexadecimal format, in
                            field 1.
           JSR TVSKIP       Skip one space after field 1.
           PLA              Restore value of currently selected byte.
           JSR VUCHAR       Display that byte, in graphic
                            form, in field 2.
           JSR TVSKIP       Skip one space after field 2.
                            Display 6502 register images in fields 4
                            thru 7:
           LDX #0
VUREGS     LDA REGS,X       Look up the register image.
           JSR VUBYTE       Display it in hexadecimal format.
           JSR TVSKIP       Skip one space after hexadecimal field.
           INX              Get ready for next register...
           CPX #4           Done 4 registers yet?
           BNE VUREGS       If not, do next one...
           RTS              If all registers displayed, return.
```

Get Currently Selected Byte

Note that the subroutine LINE.2, which puts up the second line of the Visible Monitor's display, does not itself "know" the value of the currently selected byte. Rather, it calls a subroutine, GET.SL, which returns the contents of the address pointed to by SELECT. That makes life easy for LINE.2, but how does GET.SL work?

If SELECT were a zero-page pointer, GET.SL could be a very simple subroutine and take advantage of the 6502's indirect addressing mode:

```
GET.SL   LDY #0           Get the zeroth byte above
         LDA (SELECT),Y   the address pointed to by SELECT.
         RTS              Return to caller.
```

However, SELECT is not a zero-page pointer; it's up in page $12. And the 6502 doesn't have an addressing mode that will let us load a register using any pointer not in the zero page. So how can we see what's in the address pointed to by SELECT?

We can do it in two steps. First, we'll set a zero-page pointer equal in value to the SELECT pointer, so it points to the same address; and then, since we already know how to load the accumulator using a zero-page pointer, we'll load the accumulator using the zero-page pointer that now equals SELECT. Let's call that zero-page pointer GETPTR, since it will allow us to *get* the selected byte. Using such a strategy, GET.SL can look like this:

```
GET.SL   LDA SELECT       Set GETPTR equal to
         STA GETPTR       SELECT: first the low byte;
         LDA SELECT+1     then the
         STA GETPTR+1     high byte.
         LDY #0           Get the zeroth byte above
         LDA (SELECT),Y   the address pointed to by GETPTR.
         RTS              Return to caller, with A bearing the con-
                          tents of the address specified by
                          SELECT.
```

This second attempt at GET.SL will load the accumulator with the currently selected byte, even when SELECT is not in the zero page. However, beware because by setting GETPTR equal to SELECT, GET.SL changes the value of GETPTR. This can be very dangerous. What, for example, if some other program were using GETPTR for something? That other program would be sabotaged by GET.SL's actions. If we let GET.SL change the value of GETPTR, then we must make sure that no other program ever uses GETPTR.

Such policing is hard work — and almost impossible if you want your software to run on a system in conjunction with software written by anyone else. Since I want the Visible Monitor to share your system's ROM input/output routines, and since I have no way of knowing what zero-page addresses those routines may use, I must refrain from using any of those zero-page bytes myself. When I have to use zero-page bytes — as now, so that GET.SL can use the 6502's indirect addressing mode — I must restore any zero-page bytes I've changed.

Therefore, GET.SL must be a four-part subroutine, which will: 1) save GETPTR; 2) set GETPTR equal to SELECT; 3) load the accumulator with the contents of the address pointed to by GETPTR; and finally, 4) restore GETPTR to its original value. This larger, slower, but infinitely safer version of GET.SL looks like this:

```
GET.SL    LDA GETPTR        Save GETPTR
          PHA               on stack and
          LDX GETPTR+1      in X register.

          LDA SELECT        Set GETPTR
          STA GETPTR        equal to
          LDA SELECT+1      SELECT.
          STA GETPTR+1

          LDY #0            Get the contents of the
          LDA (GETPTR),Y    byte pointed to by SELECT,
          TAY               and save it in Y register.

          PLA               Restore GETPTR
          STA GETPTR        from stack
          STX GETPTR+1      and from X register.
          TYA               Restore contents of current byte from
                            temporary storage in Y to A.
          RTS               Return with contents of currently
                            selected byte in accumulator and with
                            the zero page preserved.
```

Display Arrow Line

This routine displays an up-arrow directly underneath the current field:

LINE.3

```
LINE.3     LDX   #2          Set TV.PTR to
           LDY   #4          beginning of
           JSR   TVTOXY      arrow line.
           LDY   FIELD       Look up current field.
           SEC               If it is out of bounds,
           CPY   #7          set it to
           BCC FLD.OK        default field
           LDY #0            (the address field).
           STY FIELD
FLD.OK     LDA   FIELDS,Y    Look up column number for current
                             field.
           TAY               Use that column number as an index in-
                             to the row.
           LDA   ARROW       Load accumulator with your system's
                             graphic code for up-arrow.
           STA (TV.PTR),Y    Store up-arrow code in the Yth column
                             of the arrow line.
           RTS               Return to caller.
FIELDS     .BYTE 3,6,8       This data area shows which column
           .BYTE $0B,$0E     should get an up-arrow to indicate
           .BYTE $11,$14     any one of fields 0 thru 6. Changing one
                             of these values will cause the up-arrow
                             to appear in a different column when in-
                             dicating a given field.
```

Now that we have all the routines we need for the monitor display, let us look at how they fit together to form a structure. Here is the hierarchy of subroutines in DSPLAY:

```
MONITOR DISPLAY
        DISPLAY LABEL LINE
        DISPLAY DATA LINE
              GET.SL
              VUBYTE
                  ASCII
                  TVPLUS
              TVSKIP
        DISPLAY ARROW LINE
```

When DSPLAY is called, it will clear the top four rows of the screen, display labels, data, the arrow, and then return. How long do you think it will take to do all this? The code may look cumbersome, but the display is *quick*!

Monitor Update

The UPDATE routine is the monitor subroutine that executes functions in response to various keys. The basic key functions we want to implement are as follows:

Key	Function
GREATER THAN	Move arrow one field to the right.
LESS THAN	Move arrow one field to the left.
SPACEBAR	Increment address being displayed. (Step forward through memory.)
RETURN	Decrement address being displayed. (Step backward through memory.)

If the arrow is in fields 1, 3, 4, 5, or 6, then, for

keys 0 thru 9, A thru F	Roll a hexadecimal character into the field pointed to by the arrow.

If the arrow is under field 2 (the graphic field) then, for

All keys	Enter the key's character into field 2 (ie: enter the key's character into the displayed address).

Since the video display need not be refreshed (redisplayed within a given time) by the processor, the UPDATE routine need not return within a given amount of time. The UPDATE routine, therefore, can wait indefinitely for a new character from the keyboard, and then take appropriate action.

We can diagram these functions as shown in figure 6.5. You add additional functions to this routine by adding additional code to test the input character. You then call the appropriate function subroutine which you write.

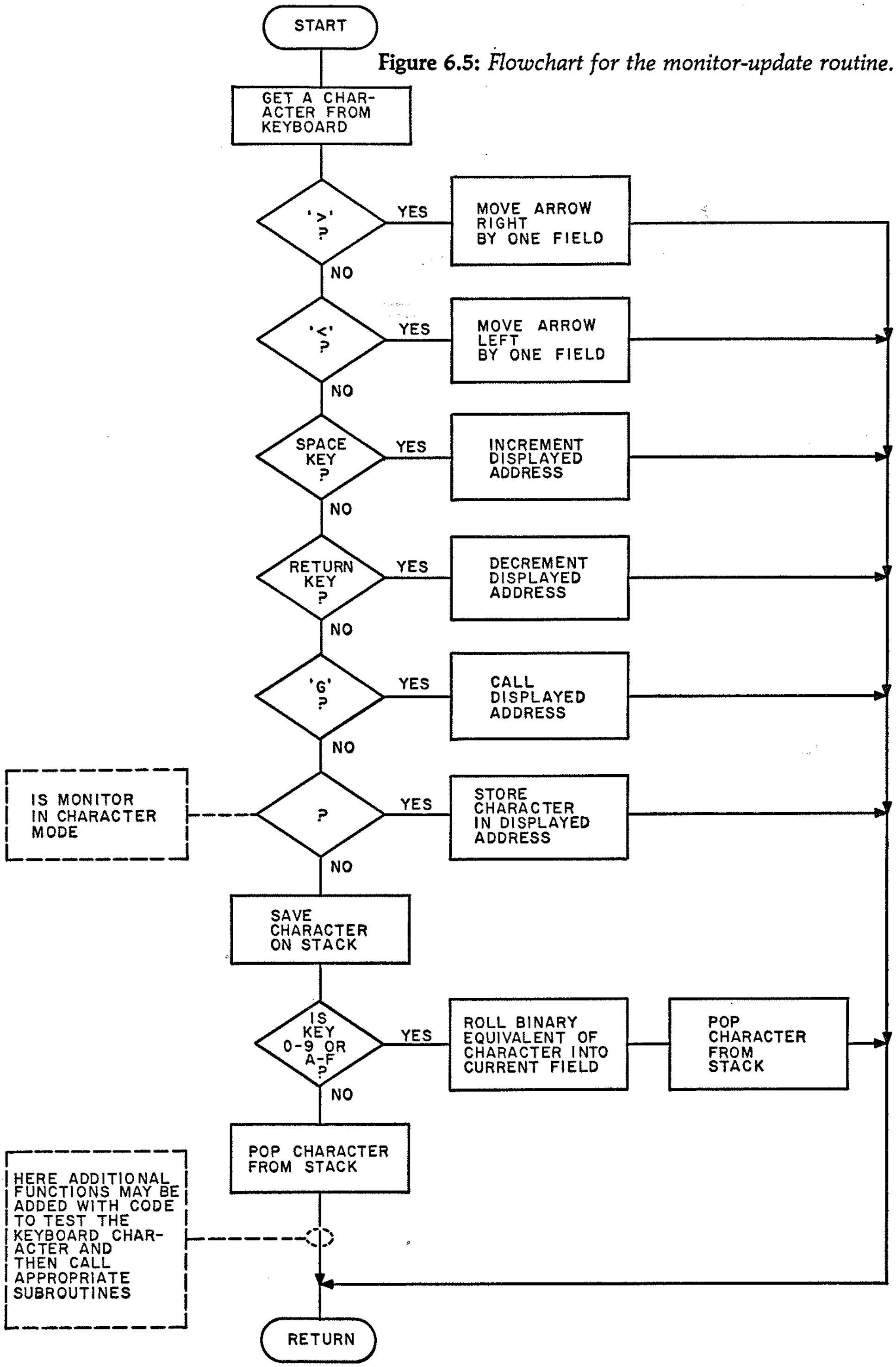

Figure 6.5: *Flowchart for the monitor-update routine.*

Get a Key

First we need a way to get a key from the keyboard. I assume that your system has a read-only memory routine to perform this function. Place the address of that routine (see the appropriate appendix for your system) into a pointer called ROMKEY located at address $1008. Once you have set the ROMKEY pointer, you can get a key by calling a subroutine labeled GETKEY, which simply transfers control to the ROM routine whose address you placed in ROMKEY:

```
GETKEY      JMP (ROMKEY)
```

Now that we have a way to get a key from the keyboard, we should be able to write source code for the monitor-update routine:

Update

```
UPDATE      JSR GETKEY       Get a character from the keyboard.
IF.GRTR     CMP #'>          Is it the GREATER THAN key?
            BNE IF.LSR       If not, perform next test.
NEXT.F      INC FIELD        If so, select the next field.
            LDA FIELD        If arrow was at the right-most field,
            CMP #7           place it underneath the left-most
            BNE EXIT.1       field.
            LDA #0
            STA FIELD
EXIT.1      RTS              Then return.
IF.LSR      CMP #'<          Is it the LESS THAN key?
            BNE IF.SP        If not, perform next test.
PREV.F      DEC FIELD        If so, select previous field:
            BPL EXIT.2       the field to the left of the
            LDA #6           current field. If arrow was at
            STA FIELD        left-most field, place it under
                             right-most field.
EXIT.2      RTS              Then return.
IF.SP       CMP #SPACE       Is it the space bar?
            BNE IF.CR        If not, perform next test.
INC.SL      INC SELECT       If so, step forward through
            BNE EXIT.3       memory, by incrementing the
            INC SELECT+1     pointer that specifies the displayed
                             address.
EXIT.3      RTS              Then return.
IF.CR       CMP #CR          Is it carriage return?
            BNE IFCHAR       If not, perform next test.
```

```
DEC.SL      LDA SELECT        If so, step backward through
            BNE NEXT.1        memory by decrementing the
            DEC SELECT+1      pointer that selects the
NEXT.1      DEC SELECT        address to be displayed.
            RTS               Then return.
IFCHAR      LDX FIELD         Is arrow underneath the
            CPX #2            character field (field 2)?
            BNE IF.GO         If not, perform next test.
                              Put the contents of A into the currently
                              selected address.
PUT.SL      TAY               Use Y to hold the character we'll put in
                              the selected address.
            LDA TV.PTR        Save zero-page pointer TV.PTR
            PHA               on stack and in X before we
            LDX TV.PTR+1      use it to put character in selected ad-
                              dress.
            LDA SELECT        Set TV.PTR equal to SELECT,
            STA TV.PTR        so it points to the
            LDA SELECT+1      currently selected
            STA TV.PTR+1      address.
            TYA               Restore to A the character we'll put in
                              the selected address.
            LDY #0            Store it in the
            STA (TV.PTR),Y    selected address.
            STX TV.PTR+1      Restore TV.PTR to
            PLA               its original value.
            STA TV.PTR
            RTS               Return to caller, with character origi-
                              nally in A now in the selected address
                              and with zero page unchanged.
            RTS               Then return.
IF.GO       CMP #'G           Is it 'G' for GO?
            BNE IF.HEX        If not, perform next test.
GO          LDY REG.Y         If so, load the 6502 registers
            LDX REG.X         with their displayed images.
            LDA REG.P
            PHA
            LDA REG.A
            PLP
            JSR CALLSL        Call the subroutine at the selected ad-
                              dress.
            PHP               When subroutine returns,
            STA REG.A         save register values in register
            STX REG.X         images.
```

```
            STY REG.Y
            PLA
            STA REG.P
            RTS              Then return to caller.
CALLSL      JMP (SELECT)     Call the subroutine at the selected ad-
                             dress.
IF.HEX      PHA              Save keyboard character.
            JSR BINARY       If accumulator holds ASCII character
                             for 0 thru 9 or A thru F, BINARY
                             returns the binary representation of that
                             hexadecimal digit. Otherwise BINARY
                             returns with A = FF and the minus flag
                             set.
            BMI OTHER        If accumulator did not hold a hexa-
                             decimal character, perform next test.
            TAY
            PLA
            TYA
ROLLIN      LDX FIELD        Roll A into a hexadecimal field.
            BNE NOTADR       Is arrow underneath the address field
                             (field 0)? If not, the arrow must be
                             under another hexadecimal field.
ADRFLD      LDX #3           Since arrow is underneath the address
LOOP.1      CLC              field, roll accumulator's hexadecimal
            ASL SELECT       digit into the address field by rolling it
            ROL SELECT+1     into the pointer that selects the
            DEX              displayed address.
            BPL LOOP.1
            TYA
            ORA SELECT
            STA SELECT
            RTS              Then return.
NOTADR      CPX #1           Is arrow underneath field 1?
            BNE REGFLD       If not, it must be underneath a register
                             image.
ROL.SL      AND #$0F         Roll A's 4 LSB into contents
            PHA              of currently selected byte.
            JSR GET.SL       Get the contents of the selected
            ASL A            address and shift left 4 times.
            ASL A
            ASL A
            ASL A
            AND #$F0
            STA TEMP         Save it in a temporary variable.
```

```
             PLA              Get original A's 4 LSB and
             ORA TEMP         OR them with shifted contents of
                              selected address.
             JSR PUT.SL       Store the result in the selected
             RTS              address and return.

TEMP         .BYTE 0          This byte holds the temporary variable
                              used by ROL.SL.
REGFLD       DEX              The arrow must be underneath a
             DEX              register image — field 3, 4, 5, or 6.
             DEX
             LDY #3
LOOP.2       CLC              Roll accumulator's hexadecimal digit
             ASL REGS,X       into appropriate register image...
             DEY
             BPL LOOP.2
             ORA REGS,X
             STA REGS,X
             RTS              ...Then return.
OTHER        PLA              Restore the raw keyboard character that
                              we saved on the stack.
             CMP#'Q           Is it 'Q' for Quit?
             BNE NOT.Q        If not, perform next test.
             PLA              If so, return to
             PLA              the caller of
             PLP
             RTS              VISMON.
NOT.Q        JSR DUMMY        Replace this call to DUMMY with a call
                              to any other subroutine that extends the
                              functionality of the Visible Monitor.
DUMMY        RTS              Return to caller.
```

ASCII to BINARY Conversion

The Visible Monitor's UPDATE subroutine requires a subroutine called BINARY, which will determine if the character in the accumulator is an ASCII 0 thru 9 or A thru F, and, if so, return the binary equivalent. On the other hand, if the accumulator does not contain an ASCII 0 thru 9 or A thru F, BINARY will return an error code, $FF. Thus:

If accumulator holds	BINARY will return
$30 (ASCII "0")	$00
$31 (ASCII "1")	$01
$32 (ASCII "2")	$02
$33 (ASCII "3")	$03
$34 (ASCII "4")	$04
$35 (ASCII "5")	$05
$36 (ASCII "6")	$06
$37 (ASCII "7")	$07
$38 (ASCII "8")	$08
$39 (ASCII "9")	$09
$41 (ASCII "A")	$0A
$42 (ASCII "B")	$0B
$43 (ASCII "C")	$0C
$44 (ASCII "D")	$0D
$45 (ASCII "E")	$0E
$46 (ASCII "F")	$0F
Any other value	$FF

We could solve this problem with a table, BINTAB, for BINary TABle. If BINTAB is at address $2000, then $2000 would contain a $FF, as would $2001, $2002, and all addresses up to $202F, because none of the ASCII codes from $00 thru $2F represent any of the characters 0 thru 9 or A thru F. On the other hand, address $2030 would contain 00, because $30 (its offset into the table) is an ASCII zero, so $2030 gets its binary equivalent: $00, a binary zero. Similarly, since $31 is an ASCII '1,' address $2031 would contain a binary '1:' $01. $2032 would contain a $02; $2033 would contain a $03, and so on up to $2039, which would contain a $09.

Addresses $203A thru $2040 would each contain $FF, because none of the ASCII codes from $3A thru $40 represent any of the characters 0 thru 9 or A thru F. On the other hand, address $2041 would contain a $0A, because $41 is an ASCII 'A' and $0A is its binary equivalent: a binary 'A.' By the same reasoning, $2042 would contain $0B; $2043 would contain $0C, and so on up to $2046, which would contain $0C, and so on up to $2046, which would contain $0F. Addresses $2047 thru $20FF would contain $FFs because none of the values $47 thru $FF is an ASCII 0 thru 9 or A thru F.

To use such a table, BINARY need only be a very simple routine:

```
BINARY    TAY               Use ASCII character as an index.
          LDA BINTAB,Y      Look up entry in BINary TABle.
          RTS               Return with it.
```

This is a typical example of a fast and simple table lookup code. But it requires a 256-byte table. Perhaps slightly more elaborate code can get by with a smaller table, or do away altogether with the need for a table. Such code must calculate, rather than look up, its answers. Let's look closely at the characters we must convert.

Legal inputs will be in the range $30 thru $39 or the range $41 thru $46. An input in the range $30 thru $39 is an ASCII 0 thru 9, and subtracting $30 from such an input will convert it to the corresponding binary value. An input in the range $41 thru $46 is an ASCII A thru F, so subtracting $36 will convert it to its corresponding binary value. For example, $41 (an ASCII 'A') minus $36 equals $0A (a binary 'A'). Any value not in either of these ranges is illegal and should cause BINARY to return a $FF.

Given these input/output relationships, BINARY need only determine whether the character in the accumulator lies in either legal range, and if so perform the appropriate subtraction, or, if the accumulator is not in a legal range, then return a $FF.

Here's some code for BINARY which makes these judgments, thus eliminating the need for a table:

```
BINARY   SEC          Prepare to subtract.
         SBC #$30     Subtract $30 from character.
         BCC BAD      If character was originally less than $30,
                      it was bad, so return $FF.
         CMP #$0A     Was character in the range $30 thru
                      $39?
         BCC GOOD     If so, it was a good input, and we've
                      already converted it to binary by sub-
                      tracting $30, so we'll return now with
                      the character's binary equivalent in the
                      accumulator.
         SBC #7       Subtract 7.
         CMP #$10     Was character originally in the range
                      $41 thru $46?
         BCS GOOD     If so, it was a good input, and we've
                      already converted it to binary by sub-
                      tracting $37, so we'll return now with
                      the character's binary equivalent in the
                      accumulator.
BAD      LDA #$FF     Indicate a bad input by returning
         RTS          minus, with A holding $FF.
GOOD     LDX #0       Indicate a good input by returning
         RTS          plus, with A holding the character's
                      binary equivalent.
```

Visible Monitor Utilities

The Visible Monitor makes the following subroutines available to external callers:

BINARY	Determine whether accumulator holds the ASCII representation for a hexadecimal digit. If so, return binary representation for that digit. If not, return an error code ($FF).
CALLSL	Call the currently selected address as a subroutine.
DEC.SL	Select previous address, by decrementing SELECT pointer.
GETKEY	Get a character from the keyboard by calling machine's read-only memory routine indirectly.
GET.SL	Get byte at currently selected address.
GO	Load registers from displayed images and call displayed address. Upon return, restore register images from registers.
INC.SL	Select next byte (increment SELECT pointer).
PUT.SL	Store accumulator at currently selected address.
VISMON	Let user give the Visible Monitor commands until user presses 'Q' to quit.

Figure 6.6 illustrates the hierarchy of the various routines of the Visible Monitor, some of which are detailed in later chapters.

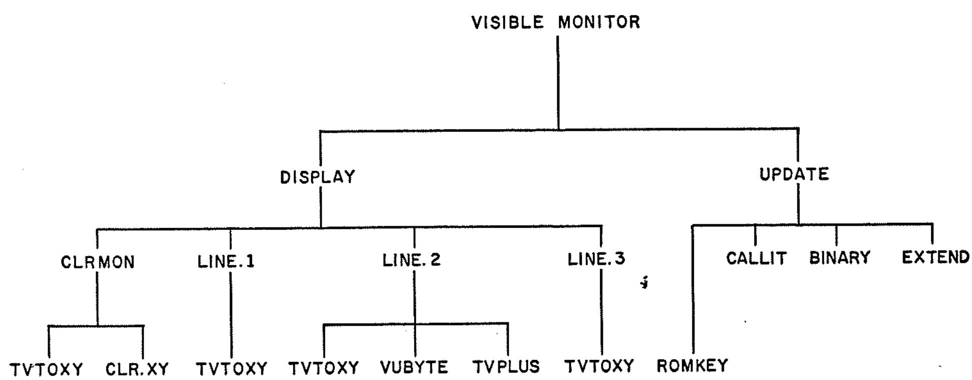

Figure 6.6: *A hierarchy of the routines of the Visible Monitor.*

Using the Visible Monitor

Use the minimal machine-language monitor on your computer to enter the Visible Monitor into memory; then have your monitor pass control to the Visible Monitor. The Visible Monitor display should appear in the upper portion of your video display. If it's not fully visible, adjust the value HOME in the screen parameters (HOME is the pointer at $1000). Use the GREATER THAN and LESS THAN character keys to move the arrow from field to field. Place the arrow under field 0 and roll hexadecimal characters into the address. Select an address in the lower portion of screen memory and use the Visible Monitor to place characters on the screen. Enter characters to the screen using both field 1 (the hexadecimal data field), and field 2 (the character field).

Select the address of the TVT routine in your system. Press G to call that subroutine. You should see the character in the accumulator print on the screen. Try exploring other memory locations. Try writing to a read-only memory address. Why doesn't that work? Try writing to the upper portion of the screen. Why doesn't that work?

Chapter 7:

Print Utilities

The Visible Monitor is a useful tool for examining and modifying memory, but at the moment it's mute: it can't "talk" to you except through the limited device of the fields in its display. You can use the Visible Monitor's character entry feature to place ASCII characters directly into screen memory, thus putting messages on the screen manually. However, as yet we have no subroutines to direct a complete message, report, or other string of characters to the screen, to a printer, or to any other output device.

Most programs require some means of directing messages to the screen, thus providing the user with the basis for informed interaction, or to a printer, thus providing a record of that interaction. This chapter presents a set of print utilities to perform these functions.

Fortunately, there are subroutines in your computer's operating system to perform character output. The Apple, Atari, OSI and PET computers each feature a routine to print a character on the screen, thus simulating a TVT (*T*ele*V*ision *T*ypewriter), and they each feature another routine to send a character to the device connected to the serial output port: usually a printer. I don't plan to reinvent those wheels in this chapter. Rather, the chapter's software will funnel all character output through code that calls the appropriate subroutine in your computer's operating system. And since we're going to have code that calls the two standard character output routines, why not provide a hook to a user-written character output routine, as well? Such a feature will make it trivial for you to direct any character output (eg: messages, hexdumps, disassembler listings, etc) to the screen and the printer, or to any special output device you may have on your system, provided that you've written a subroutine to drive that device.

Selecting Output Devices

It should be possible for any program to direct character output to the screen, and/or to the printer, and/or to the user-written subroutine. Therefore, we'll need subroutines to select and deselect (stop using) each of these devices and to select and deselect *all* of these devices. Let's call these routines TVT.ON, TVTOFF, PR.ON, PR.OFF, USR.ON, USR.OFF, ALL.ON, and ALLOFF. With these subroutines, a calling program can select or deselect output devices individually or globally.

The line of source code which will select the TVT as an output device follows:

```
JSR TVT.ON
```

This line will deselect the TVT:

```
JSR TVTOFF
```

That's a pretty straightforward calling sequence.

The select and deselect subroutines will operate on three flags: TVT, PRINTR, and USER. The TVT flag will indicate whether the screen is selected as an output device; the PRINTR flag will indicate whether the printer is selected as an output device; and the USER flag will indicate whether the user-provided subroutine is selected as an output device.

For convenience, we'll have a separate byte for each flag and define a flag as "off" when its value is zero, and "on" when its value is nonzero.

Using this definition of a flag, we can select a given device simply by storing a nonzero value in the flag for that device; we can deselect a device simply by storing a zero in the flag for that device.

The definitions for the flags and listings of the select and deselect subroutines follow:

Device Flags

	OFF = 0	When a device flag = zero, that device is not selected.
	ON = $FF	When a device flag = $FF, that device is selected.
TVT	.BYTE ON	This flag is zero if TVT is not selected; nonzero otherwise. Initially, the TVT is selected.

PRINTR	.BYTE OFF	This flag is zero if the PRINTR is not selected; nonzero otherwise. Initially, the printer is not selected.
USER	.BYTE OFF	This flag is zero if the user-provided output subroutine is not selected; nonzero otherwise. Initially, the user-provided function is deselected.

Select and Deselect Subroutines

TVT.ON	LDA #ON STA TVT RTS	Select TVT as an output device by setting the flag that indicates the "select" state of the TVT.
TVTOFF	LDA #OFF STA TVT RTS	Deselect TVT as an output device by clearing the flag that indicates the "select" state of the TVT.
PR.ON	LDA #ON STA PRINTR RTS	Select printer as an output device by setting the flag that indicates the "select" state of the printer.
PR.OFF	LDA #OFF STA PRINTR RTS	Deselect printer as an output device by clearing the flag that indicates the "select" state of the printer.
USR.ON	LDA #ON STA USER RTS	Select user-written subroutine as an output device by setting the flag that indicates the "select" state of the output routine provided by the user.
USROFF	LDA #OFF STA USER RTS	Deselect user-written subroutine as an output device by clearing the flag that indicates the "select" state of the output routine provided by the user.
ALL.ON	JSR TVT.ON JSR PR.ON JSR USR.ON RTS	Select all output devices by selecting each output device individually.
ALLOFF	JSR TVTOFF JSR PR.OFF JSR USROFF RTS	Deselect all output devices by deselecting each output device individually.

A General Character-Print Routine

Now that a calling routine can select or deselect any combination of output devices, we need a routine that will output a given character to all currently selected output devices. Let's call this routine PR.CHR, because it will *PR*int a *CHaR*acter.

All the software in this book that outputs characters will do so by calling PR.CHR; none of that software will call your system's character-output routines directly. That makes the software in this book much easier to maintain. If you ever replace your system's TVT output routine or its printer-output routine with one of your own, you won't have to change the rest of the software in this book. That software will continue to call PR.CHR. However, if many lines of code in many places called your system's character-output routines directly, then replacing a read-only memory output routine with one of your own would require you to change many operands in many places. Who needs to work that hard? Funneling all character output through one routine, PR.CHR, means we can improve our character output in the future without difficulty.

When it is called, PR.CHR will look at the TVT flag. If the TVT flag is set, it will call your system's TVT output routine. Then it will look at the PRINTR flag. If the PRINTR flag is set, it will call your system's routine that sends a character to the serial output port. Finally, it will look at the USER flag. If the USER flag is set, it will call the user-provided character-output routine. Having done all of this, PR.CHR can return. Figure 7.1 is a flowchart for PR.CHR.

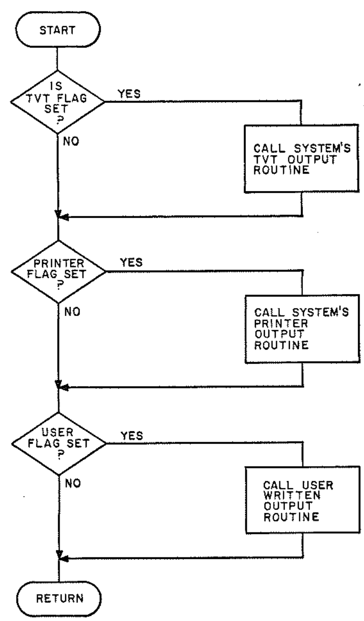

Figure 7.1: *To print a character to all currently selected output devices (PR.CHR, a general character-output routine).*

Output Vectors

If the character output routines are located at different addresses in different systems, how can PR.CHR know the addresses of the routines it must call? It can't. But it can call those subroutines indirectly, through pointers that you set.

You must set three pointers, or *output vectors*, so that they point to the character output routines in your system. A pointer called ROMTVT must point to your system's TVT output routine; a pointer called ROMPRT must point to your system's routine that sends a character to the serial output port; and a pointer called USROUT must point to your own, user-written, character-output routine. (If you have not written a special character-output subroutine, USROUT should point to a dummy routine which is nothing but an RTS instruction.) Then, if you ever relocate your TVT output routine, your printer-output routine, or your user-written output routine, you'll only have to change one output vector: ROMTVT, ROMPRT, or USROUT. Everything else in this book can remain the same.

ROMTVT, ROMPRT, and USROUT need not be located anywhere near PR.CHR. That means we can keep all the pointers and data specific to your system in one place. We can store the output vectors with the screen parameters, in a single block of memory called SYSTEM DATA. See Appendix B1, B2, B3, or B4 for your computer.

The source code of the PR.CHR routine follows:

PR.CHR

```
PR.CHR     STA CHAR      Save the character.
           BEQ EXIT      If it's a null, return without printing it.
           LDA TVT       Is TVT selected?
           BEQ IF.PR     If not, test next device.
           LDA CHAR      If so, send character indirectly to
           JSR SEND.1    system's TVT output routine.
IF.PR      LDA PRINTR    Is printer selected?
           BEQ IF.USR    If not, test next device.
           LDA CHAR      If so, send character indirectly
           JSR SEND.2    to system's printer driver.
IF.USR     LDA USER      Is user-written output subroutine
                         selected?
           BEQ EXIT      If not, test next device.
           LDA CHAR      If so, send character indirectly
           JSR SEND.3    to user-written output subroutine.
EXIT       RTS           Return to caller.
CHAR       .BYTE 0       This byte holds the last character passed
                         to PR.CHR.
```

Vectored Subroutine Calls

```
SEND.1      JMP (ROMTVT)
SEND.2      JMP (ROMPRT)
SEND.3      JMP (USROUT)
```

Specialized Character-Output Routines

Given PR.CHR, a general character-output routine, we can write specific character-output routines to perform several commonly required functions. For example, it's often necessary for a program to print a carriage return and a line feed, thus causing a new line, or to print a space, or to print a byte in hexadecimal format. Let's develop several dedicated subroutines to perform these functions. Since each of these subroutines will call PR.CHR, their output will be directed to all currently selected output devices.

Here are source listings for a few such subroutines: CR.LF, SPACE, and PR.BYT:

PRINT A CARRIAGE RETURN-LINE FEED

```
            CR = $0D          ASCII carriage return character.
            LF = $0A          ASCII line feed character.

CR.LF       LDA #CR           Send a carriage return and a
            JSR PR.CHR        line feed to the currently selected
            LDA #LF           device(s).
            JSR PR.CHR
            RTS               Return.
```

PRINT A SPACE

```
SPACE       LDA #$20          Load accumulator with ASCII space.
            JSR PR.CHR        Print it to all currently selected output
                              devices.
            RTS               Return.
```

PRINT BYTE

```
PR.BYT      PHA               Save byte.
            LSR A             Determine ASCII for the 4 MSB (most-
```

```
                    significant bits) in the
LSR A               byte:
LSR A
LSR A
JSR ASCII
JSR PR.CHR          Print that ASCII character to the current
                    device(s).
PLA                 Determine ASCII for the 4 LSB (least-
                    significant bits) in the
JSR ASCII           byte that was passed to this subroutine.
JSR PR. CHR         Print that ASCII character to the current
                    device(s).
RTS                 Return to caller.
```

Repetitive Character Output

Since some calling programs might need to output more than one space, a new line, or other character, why not have a few print utilities to perform such repetitive character outputs? In each case, the calling program need only load the X register with the desired repeat count. Then it would call SPACES to print X spaces, CR.LFS to print X new lines, or CHARS to print the character in the accumulator X times. Calling any of these routines with zero in the X register will cause no characters to be printed. To output seven spaces, a calling program would only have to include the following two lines of code:

```
LDX #7
JSR SPACES
```

To output four blank lines, a program would require these two lines of code:

```
LDX #4
JSR CR.LFS
```

To output ten asterisks, a program would need these three lines of code:

```
LDA #'*
LDX #10
JSR CHARS
```

In order to support these calling sequences, we'll need three small subroutines, SPACES, CR.LFS, and CHARS:

Print X Spaces; Print X Characters

```
SPACES      LDA #$20        Load accumulator with ASCII space.
CHARS       STX REPEAT      Initialize the repeat counter.
RPLOOP      PHA             Save character to be repeated.
            LDX REPEAT      Has repeat counter timed out yet?
            BEQ RPTEND      If so, exit. If not,
            DEC REPEAT      decrement repeat counter.
            JSR PR.CHR      Print character to all currently selected
                            output devices.
            PLA
            CLC             Loop back to repeat
            BCC RPLOOP      character, if necessary.
RPTEND      PLA             Clean up stack.
            RTS             Return to caller.
```

Print X New Lines

```
CR.LFS      STX REPEAT      Initialize repeat counter.
CRLOOP      LDX REPEAT      Exit if repeat counter has timed out.
            BEQ END.CR
            DEC REPEAT      Decrement repeat counter.
            JSR CR.LF       Print a carriage return and line feed.
            CLC             Loop back to see if done yet.
            RCC CRLOOP
END.CR      RTS             If done, return to caller.
REPEAT      .BYTE           This byte is used as a repeat counter by
                            SPACES, CHARS, and CR.LFS.
```

Print a Message

Some calling programs might need to output messages stored at arbitrary places in memory. So let's develop a subroutine, called PR.MSG, to perform this function. PR.MSG will print a message to all currently selected output devices. It must get characters from the message in a sequential manner and pass each character to PR.CHR, thus printing it on all currently selected output devices.

But how can PR.MSG know where the message starts and ends?

We could require that the message be placed in a known location, but then

PR.MSG would lose usefulness as it loses generality. We could require that a pointer in a known location be initialized so that it points to the start of the message. But that would still tie up the fixed 2 bytes occupied by that pointer. Or we could have a register specify the location of a pointer that actually points to the start of the message. Presumably a calling program can find some convenient 2 bytes in the zero page to use as a pointer, even if it must save them before it sets them. The calling program can set this zero-page pointer so that it points to the beginning of the message, and then set the X register so that it points to that zero-page pointer. Having done so, the calling program may call PR.MSG. Using the indexed indirect addressing mode, PR.MSG can then get characters from the message.

When PR.MSG has printed the entire message, it will return to its caller.

How will PR.MSG know when it has reached the end of the message? We can mark the end of each message with a special character: call it ETX, for *End* of *TeXt*. And for reasons which will become clear in Chapter 10, *A Disassembler*, we'll also start each message with another special character: TEX, for *TEX*t follows.

If we can develop PR.MSG to work from these inputs, then it won't be hard for a calling program to print any particular message in memory. Let's look at the required calling sequence.

A message, starting with a TEX and ending with an ETX, begins at some address. We'll call the high byte of that address MSG.HI and the low bye of that address MSG.LO. Thus, if the message starts at address $13A9, MSG.HI = $13 and MSG.LO = $A9.

MSGPTR is some zero-page pointer. It may be anywhere in the zero page. If the calling program does not have to preserve MSGPTR, it can print the message to the screen with the following code:

```
JSR TVT.ON         Select TVT as an output device. (Any other currently
                   selected output device will echo the screen output.)

LDA #MSG.LO        Set MSGPTR
STA MSGPTR         so it points
LDA #MSG.HI        to the start
STA MSGPTR+1       of the message.
LDX #MSGPTR        Set X register so it points to MSGPTR.
JSR PR.MSG         Print the message to all currently selected output
                   devices.
```

If the calling program must preserve MSGPTR, it will have to save MSGPTR and MSGPTR+1 before executing the above lines of code and restore MSGPTR and MSGPTR+1 after executing the above lines of code.

That looks like a reasonably convenient calling sequence. So now let's turn our attention to PR.MSG itself and develop it so it meets the demands of its callers.

Print a Message

```
PR.MSG     STX TEMP.X     Save X register, which specifies message
                          pointer.
           LDA 1,X        Save message pointer.
           PHA
           LDA 0,X
           PHA
LOOP       LDX TEMP.X     Restore original value of X, so it points
                          to message pointer.
           LDA (0,X)      Get next character from message.
           CMP #ETX       Is it the end of message indicator?
           BEQ MSGEND     If so, handle the end of the message...
           INC 0,X        If not, increment the message pointer
           BNE NEXT       so it points to the next
           INC 1,X        character in the message.
NEXT       JSR PR.CHR     Send the character to all currently
                          selected output devices.
           CLC            Get next character
           BCC LOOP       from message.
MSGEND     PLA            Restore message pointer.
           STA 0,X
           PLA
           STA 1,X
           RTS            Return to caller, with MSGPTR pre-
                          served.
TEMP.X     .BYTE 0        This data cell is used to preserve the ini-
                          tial value of X.
```

Print the Following Text

Even more convenient than PR.MSG would be a routine that doesn't require the caller to set any pointer or register in order to indicate the location of a message. But if no pointer or register indicates the start of the message, how can any subroutine know where the message starts?

It can look on the stack.

Why not have a subroutine, called Print-the-Following, which prints the message that follows the call to Print-the-Following. Since Print-the-Following is longer than six characters, let's shorten its name to "PRINT:", letting the colon in "PRINT:" suggest the phrase "the following." A calling program might then print "HELLO" with the following lines of code:

```
JSR TVT.ON          Select TVT as an output device. (Other currently
                    selected output devices will echo the screen output.)

JSR PRINT:
.BYTE TEX
.BYTE "HELLO"
.BYTE ETX
(6502 code follows the ETX)
.
.
.
```

Whenever the 6502 calls a subroutine, it pushes the address of the subroutine's caller onto the stack. This enables control to return to the caller when the subroutine ends with an RTS, because the 6502 knows it can find its return address on the stack. The subroutine PRINT: can take advantage of this fact by pulling its own return address off the stack, and using it as a pointer to the message that should be printed. When it reaches the end of the message, it can place a new return address on the stack, an address that points to the end of the message. Then PRINT: can execute an RTS. Control will then pass to the 6502 code immediately following the ETX at the end of the message. The source code for PRINT: follows:

```
PRINT:      PLA               Pull return address from
            TAX               stack and save it in
            PLA               registers X and Y.
            TAY
            JSR PUSHSL        Save the select pointer, because we're
                              going to use it as a text pointer.
            STX SELECT        Set SELECT = return address.
            STY SELECT+1
            JSR INC.SL        Increment SELECT pointer so it points
                              to TEX character.
LOOP        JSR INC.SL        Increment select pointer so it points to
                              the next character in the message.
            JSR GET.SL        Get character.
            CMP #ETX          Is it end of message indicator?
            BEQ ENDIT         If so, adjust return address and return.
            JSR PR.CHR        If not, print the character to all current-
                              ly selected devices.
            CLC               Then loop to get
            BCC LOOP          next character...
ENDIT       LDX SELECT
            LDY SELECT+1
```

```
        JSR POP.SL        Restore select pointer to its original
                          value.
        TYA               Push address
        PHA               of ETX
        TXA               onto the stack.
        PHA
        RTS               Return (to byte immediately following
                          ETX).
```

Saving and Restoring the SELECT Pointer

Now that a number of subroutines are accessing the contents of memory with the SELECT utilities (GET.SL, PUT.SL, INC.SL and DEC.SL) we should provide yet another pair of SELECT utilities to enable the subroutines to save and restore the SELECT pointer. With such save and restore functions, any subroutine can use the SELECT pointer to access memory, without interfering with the use of the SELECT pointer by other subroutines. PUSHSL will push the SELECT pointer onto the stack and POP.SL will pop the SELECT pointer off the stack. PUSHSL and POP.SL will each preserve X,Y, and the zero page.

Save Select Pointer
(Preserving X,Y, and the Zero Page)

```
PUSHSL  PLA               Pull return address from stack and
        STA RETURN        store it temporarily in RETURN.
        PLA
        STA RETURN+1
        LDA SELECT+1      Push select pointer onto stack.
        PHA
        LDA SELECT
        PHA
        LDA RETURN+1      Push return address back onto stack.
        PHA
        LDA RETURN
        PHA
        RTS               Return to caller. (Caller will find select
                          pointer on top of the stack.)
```

Restore Select Pointer (Preserving X,Y, and the Zero Page)

```
POP.SL      PLA               Save return address temporarily.
            STA RETURN
            PLA
            STA RETURN+1
            PLA               Restore select pointer from stack.
            STA SELECT
            PLA
            STA SELECT+1
            LDA RETURN+1      Place return address back on stack.
            PHA
            LDA RETURN
            PHA
            RTS               Return to caller.
RETURN      .WORD 0           This pointer is used by PUSHSL and
                              POP.SL to preserve their return ad-
                              dresses.
```

Conclusion

With the print utilities presented in this chapter, it should be easy to write the character-output portions of many programs, making it possible for calling programs to select any combination of output devices and to send individual characters, bytes, or complete messages to those devices. The calling programs will be completely insulated from the particular data representations used by the print utilities. The calling programs do not need to know the nature or location of the output-device flags or the addresses of the output vectors; they need only know the addresses of the print utilities.

Similarly, although the print utilities use subroutines that operate on the SELECT pointer, the print utilities themselves never access the SELECT pointer directly. They are completely insulated from the nature and location of the SELECT pointer. As long as they know the addresses of the SELECT utilities, the print utilities can get the currently selected byte, select the next or the previous byte, save the SELECT pointer onto the stack, and restore the SELECT pointer from the stack. If at some point we should implement a different representation of "the currently selected byte," we need only change the SELECT utilities; the print utilities, and all other programs which use the SELECT utilities need never change.

Insulating blocks of code from the internal representation of data in other blocks of code makes all the code much easier to maintain.The following print utilities are available to external callers:

CHARS	Send the character in the accumulator "X" times to all currently selected output devices.
CR.LF	Cause a new line on all currently selected devices.
CR.LFS	Cause "X" new lines on all currently selected devices.
PR.BYT	Print the byte in the accumulator, in hexadecimal representation.
PR.CHR	Print the character in the accumulator on all currently selected devices.
PR.MSG	Print the message pointed to by a zero-page pointer specified by X.
PRINT:	Print the message following the call to "PRINT:".
SPACE	Send a space to all currently selected output devices.
SPACES	Send "X" spaces to all currently selected output devices.

Exercises

1) Write a printer test program, which sends every possible character from $00 to $FF to the printer.

2) Rewrite the printer test program so that it prints just one character per line.

Chapter 8:

Two Hexdump Tools

The Visible Monitor allows you to examine memory, but only 1 byte at a time. You'll quickly feel the need for a software tool that will display or print out the contents of a whole block of memory. This is especially useful if you wish to debug a program. You can't debug a program if you're not sure what's in it. A hexdump tool will show you what you've actually entered into the computer, by displaying the contents of memory in hexadecimal form.

I've developed two kinds of hexdump programs, each for a different type of output device. When I'm working at the keyboard, I want a hexdump routine that dumps from memory to the *screen*, a line or a group of lines at a time. But for documentation and for program development or debugging away from the keyboard, I want a hexdump routine that dumps to a *printer*.

Most of the code required to dump from memory will be the same, whether we direct output to the screen or to the printer. However, there are enough differences between the two output devices that it is convenient to have two hexdump programs, one for the screen and one for the printer. Let's call them TVDUMP and PRDUMP.

TVDUMP

TVDUMP should be very responsive: when you are using the Visible Monitor, a single keystroke should cause one or more lines to be dumped to the screen. But how can TVDUMP know what lines you want to dump? Since the Visible Monitor allows you to select any address by rolling hexadecimal characters into the address field or by stepping forward and backward through memory, we might as well have

TVDUMP dump memory beginning with the currently selected address.

Since we're basing TVDUMP on the Visible Monitor's currently selected address, we can use some of the Visible Monitor's subroutines to operate on that address. GET.SL will get the currently selected byte, and INC.SL will increment the SELECT pointer, thereby selecting the next byte. The print utilities TVT.ON and PR.BYT will let us select the screen as an output device and print the accumulator in hexadecimal representation.

We ought to have TVDUMP provide a dump that will be easily readable, even on the narrow confines of a twenty-five- or forty-column display. That means we can't display a full hexadecimal line (16 bytes) on one screen line if we want to have a space between each byte. We can provide hexdumps that split each hexadecimal line into two screen lines. See outputs A and B in figure 8.1.

Output A:

```
0200   HH HH HH HH HH HH HH HH HH
0208   HH HH HH HH HH HH HH HH HH

0210   HH HH HH HH HH HH HH HH HH
0218   HH HH HH HH HH HH HH HH HH

-----------29 columns-----------
```

Output B:

```
0200
HH HH HH HH HH HH HH HH
0208
HH HH HH HH HH HH HH HH

0210
HH HH HH HH HH HH HH HH
0218
HH HH HH HH HH HH HH HH

-------23 columns-------
```

Figure 8.1: *Two TVDUMP formats.*

One way to provide such a hexdump is shown by the flowchart in figure 8.2. Using this flowchart as a guide, let's develop source code to perform the TVDUMP function:

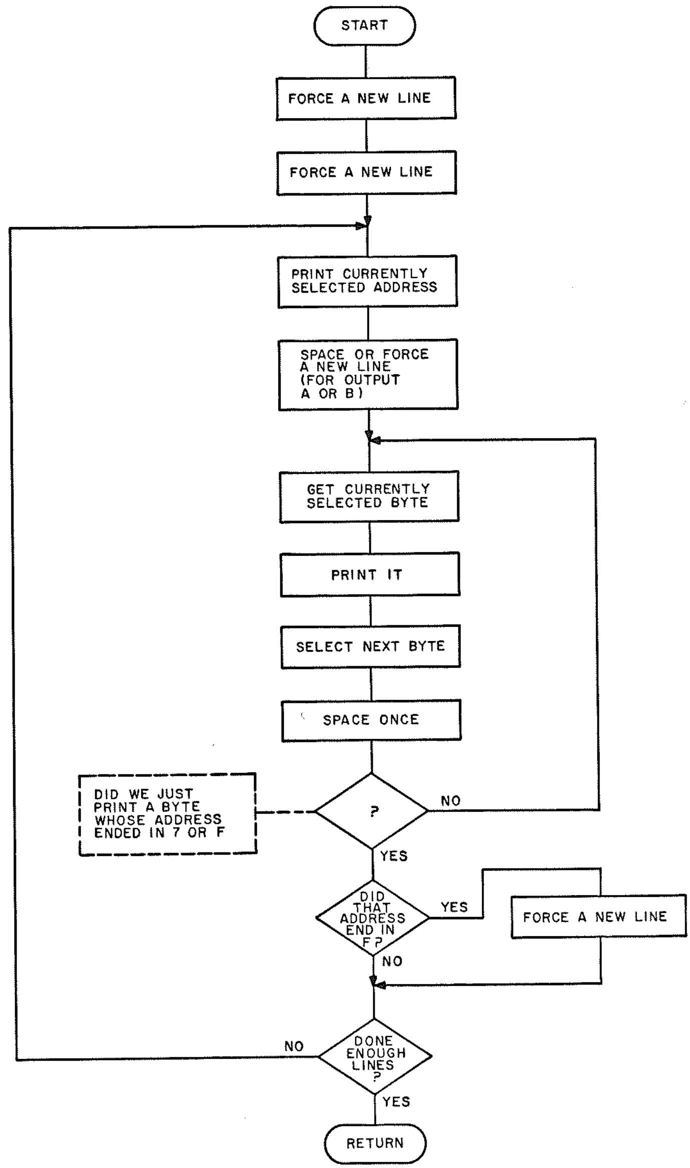

Figure 8.2: *Flowchart of the screen Hexdump Program.*

CONSTANTS

CR = $0D	Carriage return.
LF = $0A	Line feed.

REQUIRED SUBROUTINES

GET.SL	Get currently selected byte.
INC.SL	Increment the pointer that specifies the currently selected byte.
PR.BYT	Print the accumulator to currently selected devices, in hexadecimal representation.
SELECT	Pointer to currently selected address.

VARIABLES

COUNTR	.BYTE 0	This byte counts the number of lines dumped by TVDUMP.
HEXLNS	.BYTE 4	Number of hexadecimal lines to be dumped by TVDUMP. (Set this to any number you like. To dump a single hexadecimal line [16 bytes] , set HEXLNS = 1.)

TVDUMP

```
TVDUMP     JSR TVT.ON       Select TVT as an output device.
                            (Other devices will echo the dump.)
           LDA HEXLNS       Set COUNTR to the number of lines
           STA COUNTR       to be dumped by TVDUMP.
           LDA SELECT       Set SELECT to beginning
           AND #$F8         of a screen line (8 bytes)
           STA SELECT       by zeroing 3 LSB in SELECT.
           LDX #2           Skip two lines on the screen.
           JSR CR.LFS
DUMPLN     JSR PR.ADR       Print the selected address.
           JSR CR.LF        Advance to a new line on the screen.
                            (This call to CR.LF may be replaced
                            with a call to SPACE on systems with
                            screens more than 27 columns wide.
                            This will yield the Output A rather than
```

```
                JSR SPACE       Output B.)
DMPBYT          JSR SPACE       Print a space.
                JSR DUMPSL      Dump currently selected byte.
                JSR INC.SL      Select next address by incrementing
                                select pointer.
                LDA SELECT      Is it the beginning of a new
                AND #07         screen line? (3 LSB = 0?)
                BNE DMPBYT      If not, dump next byte...
                JSR CR.LF       If so, advance to a new line on the
                                screen.
                LDA SELECT      Does this address mark the beginning of
                                a new hexadecimal line?
                AND #$0F        (4 LSB of SELECT = 0?)
                BNE IFDONE
                JSR CR.LF       If so, skip a line on the screen.
IFDONE          DEC COUNTR      Dumped last line yet?
                BNE DUMPLN      If not, dump next line.
                JSR TVTOFF      Deselect TVT as an output device.
                RTS             Return to caller.
```

DUMP CURRENTLY SELECTED BYTE

This subroutine gets the currently selected byte (the byte pointed to by SELECT) and prints it in hexadecimal format on all selected devices.

```
DUMPSL          JSR GET.SL      Get currently selected byte.
                JSR PR.BYT      Print it in hexadecimal format.
                RTS             Return to caller.
```

PRINT ADDRESS

This subroutine prints, on all selected devices, the currently selected address (ie: the value of the SELECT pointer).

```
PR.ADR          LDA SELECT+1    Get the high byte of SELECT...
                JSR PR.BYT      ...and print it in hexadecimal format.
                LDA SELECT      Get the low byte of SELECT...
                JSR PR.BYT      ...and print it in hexadecimal format.
                RTS             Then return to caller.
```

PRDUMP

With the subroutine presented thus far in this chapter, we can dump to the screen just by calling TVDUMP. But what if we want to *print* a hexdump? Is a hexdump program that prints any different from one that dumps to the screen? Can we simply select the printer instead of the TVT and leave the rest of the code the same?

We could. But then we wouldn't be taking full advantage of the printer. TVDUMP produces an output that is easily read within the twenty-five or forty columns of a video display. Most printers can output sixty-four columns or more. We should take advantage of the extra width offered by a printer.

We should also recognize the difference in responsiveness between a screen and a hard-copy device. When I'm using a screen-based hexdump, I don't mind hitting a single key every time I want some lines dumped to the screen. But with a printing hexdump, I don't want to strike a key repeatedly to continue the dump. I don't mind striking a number of keys at the beginning in order to specify the memory to be dumped, but once I've done that I don't want to be bothered again. I want to set it and forget it.

When called, a printing hexdump program should announce itself by clearing the screen and displaying an appropriate title (eg: "PRINTING HEXDUMP"). Then it should ask you to specify the starting address and the ending address of the memory to be dumped.

Once it knows what you want to dump, PRDUMP should print a hexdump of the specified block of memory. For your convenience, PRDUMP should tell you what block of memory it will dump; then it should provide a header for each column of data and indicate the starting address of each line of data. (See the "D" appendices.)

Using the flowchart of figure 8.3 as a guide, we can write source code for the top level of the PRINTING HEXDUMP:

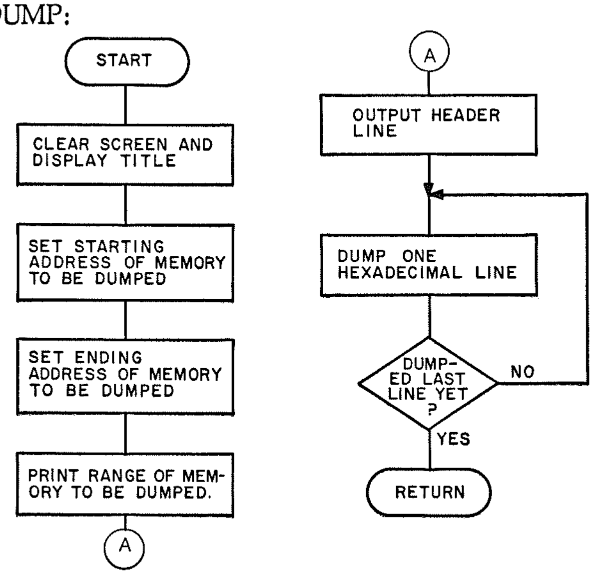

Figure 8.3: *To print a Hexdump.*

```
PRDUMP   JSR TITLE          Display the title.
         JSR SETADS         Let user set start address and end ad-
                            dress of memory to be dumped.
                            (SETADS returns with SELECT=EA,
                            the end address.)
         JSR GOTOSA         Set SELECT=SA, the starting address.
         JSR PR.ON          Select printer as a output device. (Other
                            selected devices will echo the dump.)
         JSR HEADER         Output hexdump header.
HXLOOP   JSR PRLINE         Dump one line. (PRLINE returns minus
                            if it dumped through ending address;
                            otherwise it returns PLUS.)
         BPL HXLOOP         Done yet? If not, dump next line.
         JSR CR.LF          If so, go to a new line.
         JSR PR.OFF         Deselect printer.
         RTS                Return to caller. Specified memory has
                            been dumped.
TITLE    JSR CLR.TV         Clear the screen.
         JSR TVT.ON         Select screen as an output device.
         JSR PRINT:         Display "Printing Hexdump" on all
                            selected output devices.
         .BYTE TEX          Text string must start with a TEX
                            character...
         .BYTE CR,'PRINTING '
         .BYTE 'HEXDUMP ',CR
         .BYTE LF,LF,
         .BYTE ETX          ...and end with an ETX character.
         RTS                Return to caller.
```

Get Starting, Ending Address

The printing hexdump program must secure from the user the starting address and the ending address of the memory to be dumped. The subroutine, SETADS, will perform these functions. It will place an appropriate prompt on the screen ("Set Starting Address" or "Set Ending Address") and then allow the user to specify an address.

Putting a prompt on the screen is easy: just select the TVT by calling TVT.ON, call "PRINT:" and follow this call with a TEX (start of text) character, the text of the prompt, and then an ETX (end of text) character. How can we allow the user to specify an address? We could make a subroutine, called GETADR, which gets an address by enabling the user to set some pointer. That sounds mighty familiar — that's what the Visible Monitor does. Conveniently, the Visible Monitor is a subroutine, which returns to its caller when the user presses Q for Quit. Therefore, after putting the appropriate prompt on the screen, SETADS will call the Visible Monitor. When the Visible Monitor returns, the SELECT pointer will specify the requested address.

SET STARTING ADDRESS, ENDING ADDRESS

```
SETADS    JSR TVT.ON       Select TVT as an output device. All
                           other selected output devices will echo
                           the screen output.
          JSR PRINT:       Put prompt on the screen:
          .BYTE TEX
          .BYTE CR,LF,LF
          .BYTE            'SET STARTING ADDRESS '
          .BYTE            'AND PRESS "Q".'
          .BYTE ETX
          JSR VISMON       Call the Visible Monitor, so user can
                           specify a given address.
          JSR SAHERE       Set starting address equal to address set
                           by the user.
SET.EA    JSR PRINT:       Put prompt on the screen:
          .BYTE TEX
          .BYTE CR,LF,LF
          .BYTE            'SET ENDING ADDRESS '
          .BYTE            'AND PRESS "Q".'
          .BYTE ETX
          JSR VISMON       Call the Visible Monitor, so user can
                           specify a given address.
          SEC              If user tried to set an
          LDA SELECT+1     ending address less than
          CMP SA+1         the starting address,
          BCC TOOLOW       make user do it over.
          BNE EAHERE       If SELECT is greater than SA, set
                           EA=SELECT. That will make EA
                           greater than SA.
          LDA SELECT
          CMP SA
          BCC TOOLOW
EAHERE    LDA SELECT+1     Set EA=SELECT...
          STA EA+1
          LDA SELECT
          STA EA
          RTS              ... and return.
SAHERE    LDA SELECT+1     Set SA=SELECT...
          STA SA+1
```

```
            LDA SELECT
            STA SA
            RTS              ...and return.
TOOLOW      JSR PRINT:       Since user set ending address
            .BYTE STX,       too low, print error message:
            .BYTE CR,LF,LR
            .BYTE            'ERROR! '
            .BYTE            'END ADDRESS LESS '
            .BYTE            'THAN START ADDRESS, '
            .BYTE            'WHICH IS '
            .BYTE ETX
            JSR PR.SA        Print starting address. ...and let the user
                             set
            JMP SET.EA       the ending address again.
SA          .WORD 0          Pointer to starting address of memory to
                             be dumped.
EA          .WORD $FFFF      Pointer to ending address of memory to
                             be dumped.
```

Now that the user can set the starting address and the ending address for a hexdump (or for any other program that must operate on a contiguous block of memory), we should have utilities that print out the starting address, the ending address, or the range of addresses selected by the user. If the user set $D000 as the starting address and $D333 as the ending address, we should be able to call one subroutine that prints "$D000," another that prints "$D333," and a third that prints "$D000 — $D333."

Let's call these subroutines PR.SA, to print the starting address; PR.EA, to print the ending address; and RANGE, to print the range of addresses.

Print Starting Address

The following subroutine prints the value of SA, the starting address, in hexadecimal format:

```
PR.SA       LDA #'$          Print a dollar sign to
            JSR PR.CHR       indicate hexadecimal.
            LDA SA+1         Print high byte of starting address.
            JSR PR.BYT
            LDA SA           Print low byte of starting address.
            JSR PR.BYT
            RTS              Return to caller.
```

Print Ending Address

The following subroutine prints the value of EA, the ending address, in hexadecimal format:

```
PR.EA     LDA #'$        Print a dollar sign to
          JSR PR.CHR     indicate hexadecimal.
          LDA EA+1       Print high byte of ending address.
          JSR PR.BYT
          LDA EA         Print low byte of ending address.
          JSR PR.BYT
          RTS            Return to caller.
```

Print Range of Addresses

```
RANGE     JSR PR.SA      Print starting address.
          LDA #'-        Print a hyphen.
          JSR PR.CHR
          JSR PR.EA      Print ending address.
          RTS            Return to caller.
```

HEADER

We want a routine to print an appropriate header for the hexdump. It should accomplish two tasks: identify the block it will dump, and print a hexadecimal digit at the top of every column of hexdump output. Thus, HEADER should produce the output shown between the following lines:

```
DUMPING HHHH-HHHH
          0 1 2 3 4 5 6 7 8 9 A B C D E F

```

Notice the blank line following the line of hexadecimal characters. This will insure a blank line between the header and the dump itself, making for a more readable output. (See the hexdumps in the D series of appendices which were produced with PRDUMP.)

Here are a few lines of code to print the first line of the header:

```
JSR PRINT:
.BYTE TEX,CR,LF
.BYTE 'DUMPING '
.BYTE ETX
JSR RANGE
JSR CR.LF
```

What about the rest of the header? Since all we want to do is print the hexadecimal digits 0 thru $F, with appropriate spacing between them, the rest of HEADER can just be some code to count from 0 to $F, convert to ASCII, and print:

PRINT HEXADECIMAL DIGITS (Version 1)

```
            LDX #7          Print seven spaces.
            JSR SPACES
            LDA #0          Initialize column counter
            STA COLUMN      to zero.
HXLOOP      LDA COLUMN      Convert column counter to
            JSR ASCII       an ASCII character and
            JSR PR.CHR      print it.
            LDX #2          Space twice after the character.
            JSR SPACES
            INC COLUMN      Increment the column counter.
            LDA COLUMN      Loop if counter not greater
            AND #$F0        than $0F.
            BEQ HXLOOP
            LDX #2          Otherwise, skip two lines
            JSR CR.LFS      after the header.
            RTS             Then return.
COLUMN      .BYTE 0         This 1-byte variable is used to count
                            from 00 to $0F.
```

Version 1 of PRINT HEXADECIMAL DIGITS will work, and in only 49 bytes. But that's 49 bytes of code, which among other things must count and branch, and if for some reason one of those bytes is wrong, Version 1 of PRINT HEXADECIMAL DIGITS will probably go directly into outer space. But we could write PRINT

HEXADECIMAL DIGITS in a much more straightforward manner, which, though somewhat more costly in terms of memory required, will be more readable and less likely to run amuck.

PRINT HEXADECIMAL DIGITS need only call "PRINT:", and follow this call with a text string consisting of the desired hexadecimal digits.

PRINT HEXADECIMAL DIGITS (Version 2)

```
JSR PRINT:
.BYTE TEX
.BYTE '          0   1   2   3   4   5   6   7   '
.BYTE           '8   9   A   B   C   D   E   F'
.BYTE CR,LF,LF
.BYTE ETX
RTS
```

Version 2 of PRINT HEXADECIMAL DIGITS requires 60 bytes. But it's more readable than Version 1 of PRINT HEXADECIMAL DIGITS, and it can be modified much more easily: just change the text in the message it prints. You don't have to calculate branch addresses or test the terminal condition in a loop. This is just one example of a programming problem that may be solved in a computation-intensive or a data-intensive manner.

Where other factors are about equal, I prefer data-intensive subroutines, because they're more readable and easier to change. Even in this case, I'm willing to pay the extra 20 bytes for a version of PRINT HEXADECIMAL DIGITS that I don't have to read twice. Hence, PRINT HEXADECIMAL DIGITS Version 2, and not Version 1, will appear in the assembler listings of HEADER in Appendix C5.

PRLINE

Clearly, most of the work of PRDUMP will be performed by the subroutine PRLINE, which dumps one line of memory to the printer. It will stop when it has dumped 16 bytes (one hexadecimal line) or has dumped through the ending address specified by the user.

As we did for TVDUMP, let's use SELECT as a pointer to the first byte that must be dumped by PRLINE. When PRLINE is called, it must see if the currently selected byte (the byte pointed to by SELECT) is at the start of a hexadecimal line. A byte is at the beginning of a hexadecimal line if the 4 LSB (least-significant bits) of its address are zero. Thus, $4ED8 is not the start of a hexadecimal line, but $4ED0 *is*.

If the currently selected byte is not the beginning of a hexadecimal line, PRLINE should space over to the appropriate column for that byte. If the currently selected byte is at the beginning of a hexadecimal line, PRLINE should print the address of the currently selected byte and space twice.

Once it has spaced over to the proper column, PRLINE need only get the currently selected byte, print it in hexadecimal format, space once, and then do the same for the next byte, until it has dumped the entire line or has dumped the last byte requested by the user.

Figure 8.4 gives a flowchart for the following routine:

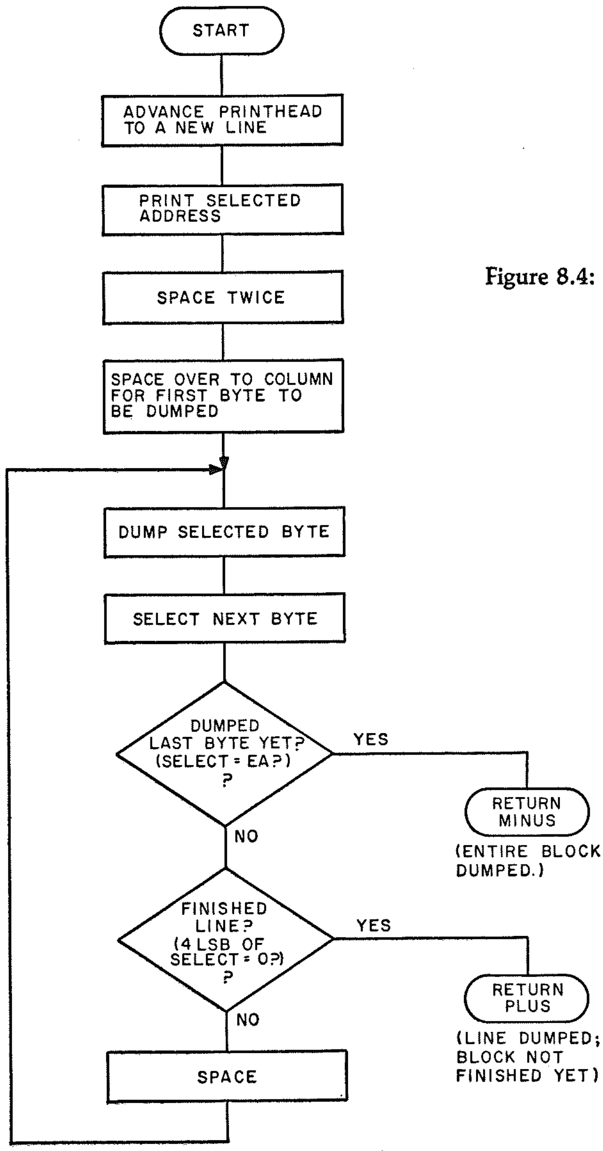

Figure 8.4: *Dump one line to the printer.*

PRLINE

```
PRLINE    JSR CR.LF      Advance printhead to a new line.
          LDA SELECT     Determine starting
          PHA            column
          AND #$0F       for this dump.
          STA COLUMN     Now COLUMN holds the number of the
                         column in which we will dump the first
                         byte.
          PLA            Set SELECT pointer to
          AND #$F0       beginning of a hexadecimal line.
          STA SELECT
          JSR PR.ADR     Print the selected address.
          LDX #3         Space three times — to the
          JSR SPACES     first column.
          LDA COLUMN     Do we dump from the first column?
          BEQ COL.OK     If so, we're at the correct column now.
LOOP      LDX #3         If not, space three
          JSR SPACES     times for each byte not
          JSR INC.SL     dumped.
          DEC COLUMN
          BNE LOOP
COL.OK    JSR DUMPSL     Dump the currently selected byte.
          JSR SPACE      Space once.
          JSR NEXTSL     Select the next byte in memory, unless
                         we've already dumped through the end
                         address.
          BMI EXIT       (MINUS means we've dumped through
                         the end address.)
NOT.EA    LDA SELECT     Dumped entire line?
          AND #$0F       (4 LSB of SELECT = 0?)
          CMP #0         If so, we've dumped the entire line. If
                         not,
          BNE COL.OK     select the next byte and dump it...
EXIT      RTS            PRLINE returns MINUS, with A=$FF,
                         if it dumped through ending address.
                         Otherwise it returns PLUS, with A=0.
```

Select Next Byte

NEXTSL tests to see if SELECT is less than the ending address. If so, it increments SELECT and returns PLUS (with zero in the accumulator). If not, it preserves SELECT and returns MINUS (with $FF in the accumulator).

NEXTSL

```
NEXTSL   SEC               Prepare to compare.
         LDA SELECT+1      Is high byte of SELECT less than
         CMP EA+1          high byte of end address (EA)?
         BCC SL.OK         If so, SELECT is less than EA, so it may
                           be incremented.
         BNE NO.INC        If SELECT is greater than EA, don't
                           increment SELECT.
                           SELECT is in the same page as EA,
         SEC               prepare to compare low bytes:
         LDA SELECT        Is low byte of SELECT less than
         CMP EA            low byte of EA?
         BCS NO.INC        If not, don't increment it.
SL.OK    JSR INC.SL        Since SELECT is less than EA, we may
                           increment it.
         LDA #0            Set "incremented" return code and
         RTS               return.
NO.INC   LDA #$FF          Set "not incremented" return code
         RTS               and return.
```

Go to Start of Block

GOTOSA sets SELECT = SA, thus selecting the first byte in the block defined by SA and EA:

```
GOTOSA   LDA SA            Set SELECT
         STA SELECT        equal to
         LDA SA+1          START ADDRESS
         STA SELECT+1      of block.
         RTS
```

Now the two hexdump tools are complete. You may invoke either tool directly from the Visible Monitor by displaying the start address of the given hexdump tool and pressing "G." This will work fine for PRDUMP: you'll get a chance to set the starting address and the ending address that you want to dump, and then you'll see the dump on both the printer and the screen. If you start TVDUMP with a "G" from the Visible Monitor, you'll only get a dump of TVDUMP itself. You won't be able to use TVDUMP to dump any other location in memory. Why? Because TVDUMP dumps from the displayed address, and to start any program with a "G" from the Visible Monitor, you must first display the starting address of that program. Probably you'd like to be able to use TVDUMP to dump other areas in memory. To do so, you must assign a Visible Monitor key (eg: "H") to the subroutine TVDUMP, so that the Visible Monitor will call TVDUMP whenever you press that key. See Chapter 12, *Extending the Visible Monitor.*

Chapter 9:

A Table-Driven Disassembler

With the Visible Monitor you can enter object code into your computer. With hexdump tools you can dump that object code to the screen or to a printer. However, you still can't be sure you've entered the instructions you intended to enter unless you refer back and forth from your hexdump to Appendix A4, *The 6502 Opcode List*. You must verify that every opcode you entered is for the instruction and the addressing mode that you had intended. You must count forward or backward in hexadecimal to make sure that the operands in your branch instructions are correct. If you entered one opcode or operand incorrectly, then even though your handwritten program may be correct, the version in your computer's memory will be wrong.

A disassembler (the opposite of an assembler) can make your life a lot easier by displaying or printing the mnemonics represented by the opcodes you entered into your computer, and by showing you the actual addresses and addressing modes represented by your operands. The disassembler can't know that address 0000 has the label "TV.PTR," but it can let you know that a given instruction operates on address 0000.

A disassembled line includes the following fields:

Field Number	**Field Description**
1.	Mnemonic.
2.	Operand.
3.	Address of opcode.
4.	Opcode in hexadecimal.

5. First byte of operand (if present) in hexadecimal.
6. Second byte of operand (if present) in hexadecimal.

Here's a disassembled line, with each of the fields numbered:

1	*2*	*3*	*4*	*5*	*6*	(Field Numbers)
JSR	0400	08AC	20	00	04	(Disassembled Line)

As with hexdump tools, I find it convenient to have two disassemblers: one for the screen and one for the printer. The screen-oriented disassembler should direct a certain number of disassembled lines to the screen whenever it is called. On the other hand, the printing disassembler should get a starting address and an ending address from the user and print a continuous disassembly of that portion of memory. As before, when I direct output to a printer I want to set it and forget it.

Whether we disassemble to the screen or to a printer, we will disassemble one line at a time. How can a program disassemble a line? The same way a person does. You look at an opcode in memory and then consult a table such as Appendix A4 to determine the operation represented by that opcode. Each operation has two attributes, a mnemonic and an addressing mode. The procedure is simple. Write the mnemonic; then, from the addressing mode determine whether this opcode takes no operand, a 1-byte operand, or a 2-byte operand. If it takes an operand, look at the next byte or two in memory and then write the operand for the mnemonic.

Thus, if you wish to disassemble object code from some place in memory, and you find an $8D at that location, you can determine from Appendix A6 that $8D represents "store accumulator, absolute mode." Therefore, you'll write: "STA," which is the mnemonic for store the accumulator.

The absolute mode requires a 2-byte operand, so you'll look at the 2 bytes following the $8D. If $36 follows the $8D and is itself followed by $D0, then the disassembled line will look like this:

```
STA $D036
```

That's a lot easier to read than the original 3 bytes of object code:

```
8D 36 D0
```

DISASSEMBLY

JSR	0400	1E00	20	00	04
JSR	04A0	1E03	20	A0	04
LDA	(0021),Y	1E06	B1	21	
CLC		1E08	18		
BCC	1E00	1E09	90	F5	

HEXDUMP

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1E00	20	00	04	20	A0	04	B1	21	18	90	F5					

Figure 9.1: *Disassembly and hexdump of the same object code.*

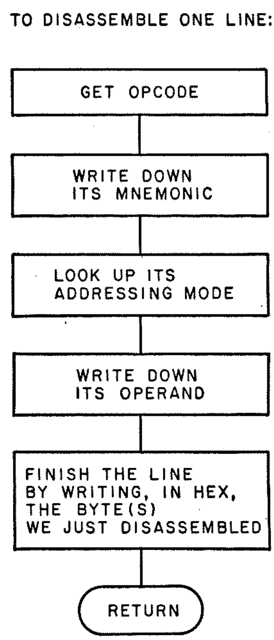

Figure 9.2: *Algorithm for disassembling one line of code.*

That looks pretty simple. We can use the SELECT pointer to indicate the current byte within memory, and we'll assume that lower-level subroutines exist or will exist to do the jobs required by DSLINE, which disassembles one line. With those assumptions, we can write source code for DSLINE:

DISASSEMBLE ONE LINE

```
DSLINE    JSR GET.SL      Get currently selected byte.
          PHA             Save it on stack.
          JSR MNEMON      Print the mnemonic represented by that
                          opcode.
          JSR SPACE       Space once.
          PLA             Restore opcode to accumulator.
          JSR OPERND      Print the operand required by that op-
                          code.
          JSR FINISH      Finish the line by printing fields 3 thru
                          6.
          JSR NEXTSL      Select next byte.
          RTS             Return to caller, with SELECT pointing
                          at the last byte of the operand (or at the
                          opcode, if it was a 1-byte instruction).
```

Print Mnemonic

We need a subroutine called MNEMON which prints the three-letter mnemonic for a given opcode. How can MNEMON do this? How do we do it? We look it up in a table such as Appendix A4. We could have a similar table in memory and then have MNEMON sequentially look up from the table the three characters comprising the desired mnemonic. That would require a 3-byte mnemonic for each of 256 possible opcodes: a 758-byte table. That's a lot of memory! Perhaps if we organize our data better we'll need less memory.

For example, why include the same mnemonic more than once in the table? Eight different opcodes use the mnemonic LDA; why should I use up 24 bytes to store "LDA" eight times? We could have a table of mnemonic names, which is nothing more than an alphabetical list of the three-letter mnemonics. There are only fifty-six different mnemonics; if we add one pseudo-mnemonic, "BAD," to mean that a given opcode is not valid, then we still have only fifty-seven mnemonics. The table of mnemonic names will therefore require only 171 bytes.

If you have a given opcode, how can you know which mnemonic in the table of mnemonic names corresponds to your opcode? A mnemonic *code* is some number that uniquely identifies a given mnemonic. Let's assume that we have a table of mnemonic codes which gives the mnemonic code for each possible opcode.

Now you can look up in the table of mnemonic codes the mnemonic code corresponding to a given opcode, and then use the mnemonic code as an index to the table of mnemonic *names*. The three sequential characters located in the table of mnemonic names will comprise the mnemonic for your original opcode.

This method requires not one but two tables. The two together, however, require considerably less memory than our first table did. The table of mnemonic codes will be 256-bytes long, since it must have an entry for every possible opcode, including invalid ones. The table of mnemonic names, on the other hand, will be only 171-bytes long, so the two tables together require only 427 bytes. That's 331 bytes or 43 percent less memory than our first table required.

Space saved in tables may not be worth it if large or complicated code is required as an index to those tables, but in this case the code is quite simple:

```
MNEMON    LDX #3          There are three letters in a mnemonic.
          STX LETTER      We'll keep track of the letters by count-
                          ing down to zero.
          TAX             Prepare to use the opcode as an index.
          LDA MCODES,X    Look up the mnemonic code for that op-
                          code. (MCODES is the table of
                          mnemonic codes.)
          TAX             Prepare to use that mnemonic code as
                          an index.
MNLOOP    LDA MNAMES,X    Get a mnemonic character. (MNAMES
                          is the list of mnemonic names.)
          STX TEMP.X      Save X register (since printing will
                          almost certainly change the X register).
          JSR PR.CHR      Print the character to all currently
                          selected devices.
          LDX TEMP.X      Restore X register to its previous value.
          INX             Adjust index for next letter.
          DEC LETTER      If three letters not yet printed,
          BNE MNLOOP      loop back to handle the next one.
          RTS             Otherwise, return to caller.
TEMP.X    .BYTE 0
LETTER    .BYTE 0
```

As you can see, MNEMON requires only 30 bytes of code in machine language: 2 bytes to hold variables and 427 bytes for the two tables (MNAMES and MCODES). The entire subroutine requires 459 bytes, but since most of those bytes are data in tables, comparatively little can go wrong with the program. If the wrong bytes are keyed into the table of mnemonic names, then the disassembler will print one or more incorrect characters in a mnemonic. But MNEMON won't crash! Bad data in means bad data out, but at least MNEMON will run, and a running program is a lot easier to correct than one that crashes and burns.

So again we have a data-intensive, rather than a computation-intensive, subroutine. The tables required by MNEMON are included in Appendix C8.

Print Operand

Now we come to the tricky part: printing the right operand given an opcode at some location in memory. When I disassemble object code by hand, I write the operand in two steps: first I determine the addressing mode of the given opcode, and then, if that addressing mode takes an operand, I write down the proper operand in the proper form. Proper form means including a comma and an X or a Y for every indexed instruction, including parentheses in the proper places for indirect instructions, and printing out all addresses *high* byte first, since that makes it easier to read an address.

OPERND (the subroutine that prints an operand for a given opcode in a given location in memory) will therefore determine the addressing mode for a given opcode, and then call an appropriate subroutine to handle that addressing mode:

OPERND

```
OPERND      TAX             Look up addressing mode code for
            LDA MODES,X     this opcode.
            TAX             X now indicates the addressing mode.
            JSR MODE.X      Call the subroutine that handles address-
                            ing mode "X."
            RTS             Return to caller.
```

MODES is a table giving the addressing mode for each opcode.

Note that OPERND can work only if we have a routine called MODE.X which somehow transfers control to the subroutine that handles addressing mode "X." How can MODE.X do this? One way is to have a table of pointers, in which the Xth pointer points to the subroutine that handles addressing mode "X." MODE.X must then transfer control to the Xth subroutine in this table. It would be nice if the 6502 offered an indexed JSR instruction, which would call the subroutine whose address is the Xth entry in the table. Unfortunately, the 6502 doesn't offer an indexed JSR instruction, so we'll have to simulate one in software.

Fortunately, the 6502 does offer an indirect JMP. If a pointer, called SUBPTR, can be made to point to a given subroutine, then the instruction JMP (SUBPTR) will transfer control to that subroutine. Therefore, MODE.X need only set SUBPTR equal to the Xth pointer in a table of subroutine pointers, and with the instruction

JMP (SUBPTR), it can transfer control to the Xth subroutine in the table.

HANDLE ADDRESSING MODE "X"

MODE.X	LDA SUBS,X	Get low byte of Xth pointer in the table of subroutine pointers.
	STA SUBPTR	Set low byte of subroutine pointer.
	INX	Adjust index to get next byte.
	LDA SUBS,X	Get high byte of Xth pointer in the table of subroutine pointers.
	STA SUBPTR+1	Set high byte of subroutine pointer.
	JMP (SUBPTR)	Jump to the subroutine specified by the subroutine pointer. That subroutine will then return to the *caller* of MODE.X, not to MODE.X itself.
SUBS		This is a table of pointers, in which the Xth pointer points to the subroutine tha handles addressing mode X.

Disassembler Utilities

Given MODE.X, OPERND can call the right subroutine to handle any give addressing mode. Now all we need are thirteen different subroutines, one for each c the 6502's different addressing modes.

Before writing those subroutines, however, let's think for a moment about wha they must do, and see if we can't write a few utility subroutines to perform thos functions. With a proper set of utilities, the addressing mode subroutines themselve need only call the right utilities in the right order.

The following set of utilities seems reasonable:

- ONEBYT: Print a 1-byte operand.
- TWOBYT: Print a 2-byte operand.
- RPAREN: Print a right parenthesis.
- LPAREN: Print a left parenthesis.
- XINDEX: Print a comma and then the letter "X."
- YINDEX: Print a comma and then the letter "Y."

Print a 1-Byte Operand: ONEBYT

```
ONEBYT      JSR INC.SL      Advance to byte following opcode.
            JSR DUMPSL      Print it in hexadecimal.
            RTS             Return to caller.
```

Print a 2-Byte Operand: TWOBYT

A 2-byte operand always specifies an address with the low byte first. To print a 2-byte operand high byte first, we must first print the second byte in the operand and *then* print the first byte in the operand; each, of course, in hexadecimal format.

```
TWOBYT      JSR INC.SL      Advance to first byte of operand.
            LDA GET.SL      Load that byte into accumulator.
            PHA             Save it.
            JSR INC.SL      Advance to second byte of operand.
            JSR DUMPSL      Print it in hexadecimal format.
            PLA             Restore the operand's first byte to the
            JSR PR.BYT      accumulator, and print it in hexa-
                            decimal.
            RTS             Return to caller.
```

ONEBYT and TWOBYT each leave SELECT pointing at the last byte of the operand.

Print Right, Left Parenthesis: RPAREN, LPAREN

RPAREN prints a right parenthesis to all currently selected devices. LPAREN prints a left parenthesis to all currently selected devices.

```
RPAREN      LDA #')         Load accumulator with ASCII code for
                            right parenthesis.
            BNE SENDIT      Send it to all currently selected devices.
LPAREN      LDA #'(         Load accumulator with ASCII code for
                            left parenthesis.
SENDIT      JSR PR.CHR      Send it to all currently selected devices.
            RTS             Return to caller.
```

Index with Register X: XINDEX

XINDEX prints a comma and then the letter "X:"

```
XINDEX       LDA #',          Load accumulator with ASCII code for a
                              comma; then print it to
             JSR PR.CHR       all currently selected devices.
             LDA #'X          Load accumulator with ASCII code for
                              the letter "X;" then print it
             JSR PR.CHR       to all currently selected devices.
             RTS              Return to caller.
```

Index with Register Y: YINDEX

YINDEX prints a comma and then the letter "Y:"

```
YINDEX       LDA #',          Load accumulator with ASCII code for a
                              comma; then print it to all
             JSR PR.CHR       currently selected devices.
             LDA #'Y          Load accumulator with ASCII code for
                              the letter "Y;" then print it
             JSR PR.CHR       to all currently selected devices.
             RTS              Return to caller.
```

So much for the disassembler utilities. Now with a single subroutine call we can print a 1-byte or a 2-byte operand (and, of course, we can print a no-byte operand), and we can print any of the frequently used characters and character combinations. Okay, let's write some addressing mode subroutines:

Addressing Mode Subroutines

Because the 6502 has thirteen different addressing modes, we'll need thirteen different addressing mode subroutines:

Subroutine	Addressing Mode
ABSLUT	Absolute

ABS.X	Absolute,X
ABS.Y	Absolute,Y
ACC	Accumulator
IMPLID	Implied
IMMEDT	Immediate
INDRCT	Indirect
IND.X	Indirect,X
IND.Y	Indirect,Y
RELATV	Relative
ZEROPG	Zero Page
ZERO.X	Zero Page,X
ZERO.Y	Zero Page,Y

The main job for each subroutine will be to print the operand in the proper form. Although a given addressing mode will always have the same number of characters in its operand, unfortunately, different addressing modes may have operands of different lengths. For example, implied addressing mode has no characters in its operand, whereas indirect indexed addressing requires eight characters in its operand, if leading zeros are included.

But no matter how many characters appear in an operand, we want to make sure that field 3 (the address field) always begins at the same column. Therefore, every addressing-mode subroutine will return with A holding the number of characters in the operand, with X holding the number of bytes in the operand, and with SELECT pointing at the last byte in the operand (or at the opcode, if it was a 1-byte instruction). Then FINISH can print an appropriate number of spaces before printing fields 3 thru 6.

Absolute Mode: ABSLUT

To print the operand for an instruction in the absolute mode, we need only print a 2-byte operand. Thus, 8D B2 04 will disassemble as:

```
STA 04B2   8D B2 04
```

```
ABSLUT      JSR TWOBYT
            LDX #2          X holds number of bytes in operand.
            LDA #4          A holds number of characters in
                            operand.
            RTS
```

Absolute, X Mode: ABS.X

To print the operand for an instruction in the absolute, X mode, we must print a 2-byte operand, a comma, and then an "X:"

LDA D09A,X   BD 9A D0

```
ABS.X      JSR ABSLUT      Print the 2-byte operand.
           JSR XINDEX      Print the comma and the "X."
           LDX #2          X holds number of bytes in operand.
           LDA #6          A holds number of characters in
                           operand.
           RTS             Return to caller.
```

Abolute, Y Mode: ABS.Y

To print the operand for an instruction in the absolute, Y mode, we must print a 2-byte operand, a comma, and then a "Y:"

ORA 02FE,Y   19 FE 02

```
ABS.Y      JSR ABSLUT      Print the 2-byte operand.
           JSR YINDEX      Print the comma and the "Y."
           LDX #2          X holds number of bytes in operand.
           LDA #6          A holds number of characters in
                           operand.
           RTS             Return to caller.
```

Accumulator Mode: ACC

To print the operand for an instruction in the accumulator mode, we need only print the letter "A:"

ROR A   6A

```
ACC         LDA #'A          Load accumulator with ASCII code for
                             the letter A.
            JSR PR.CHR       Print it on all currently selected devices.
            LDX #0           X holds number of bytes in operand.
            LDA #1           A holds number of characters in
                             operand.
            RTS              Return to caller.
```

Implied Mode: IMPLID

Implied mode has no operand, so just return:

```
CLC   18
```

```
IMPLID      LDX #0           X holds number of bytes in operand.
            LDA #0           A holds number of characters in
                             operand.
            RTS
```

Immediate Mode: IMMEDT

Immediate mode requires a 1-byte operand, which we'll print in hexadecimal format. Thus, it should disassemble the two consecutive bytes "A9 41" as follows:

```
LDA #$41   A9 41
```

```
IMMEDT      LDA #'#          Print a '#' sign.
            JSR PR.CHR
            LDA #'$          Print a dollar sign.
            JSR PR.CHR
            JSR ONEBYT       Print 1-byte operand in hexadecimal for-
                             mat.
            LDX #1           X holds number of bytes in operand.
            LDA #4           A holds number of characters in
                             operand.
            RTS              Return to caller.
```

Indirect Mode: INDRCT

To print the operand for an instruction in the indirect mode, we need only print an absolute operand within parentheses. Thus, the three consecutive bytes "6C 00 04" will disassemble as:

JMP (0400)   6C 00 04

```
INDRCT    JSR LPAREN    Print left parenthesis.
          JSR ABSLUT    Print the 2-byte operand.
          JSR RPAREN    Print the right parenthesis.
          LDX #2        X holds number of bytes in operand.
          LDA #6        A holds number of characters in
                        operand.
          RTS           Return to caller.
```

Indirect, X Mode: IND.X

To print the operand for an instruction in the indirect, X addressing mode, we need to print a left parenthesis, a zero-page address, a comma, the letter "X," and then a right parenthesis. Thus, the two consecutive bytes "A1 3C" will disassemble as:

LDA (3C,X)   A1 3C

```
IND.X     JSR LPAREN    Print a left parenthesis.
          JSR ZERO.X    Print a zero-page address, a comma, and
                        the letter "X."
          JSR RPAREN    Print a right parenthesis.
          LDX #1        X holds number of bytes in operand.
          LDA #8        A holds number of characters in
                        operand.
          RTS           Return to caller.
```

Indirect, Y Mode: IND.Y

To print the operand for an instruction in the indirect, Y mode, we must print a left parenthesis, a zero-page address, a right parenthesis, a comma, and then the letter "Y." Thus, the two consecutive bytes "B1 AF" will disassemble as:

```
LDA (AF),Y   B1 AF
```

```
IND.Y       JSR LPAREN     Print a left parenthesis.
            JSR ZEROPG     Print a zero-page address.
            JSR RPAREN     Print a right parenthesis.
            JSR YINDEX     Print a comma and then the letter "Y."
            LDX #1         X holds number of bytes in operand.
            LDA #8         A holds number of characters in
                           operand.
            RTS            Return to caller.
```

Relative Mode: RELATV

Relative mode can be tricky. A relative branch instruction specifies a forward branch if its operand is *plus* (in the range of 00 to $7F), but it specifies a backward branch if its operand is *minus* (in the range of $80 to $FF). Therefore, in order to determine the address specified by a relative branch instruction, we must first determine whether the operand is plus or minus, so we can determine whether we're branching forward or backward. Then we must add or subtract the least-significant 7 bits of the operand to or from the address immediately following the operand of the branch instruction; the result of that calculation will be the actual address specified by the branch instruction.

```
RELATV      JSR INC.SL     Select next byte in memory.
            JSR PUSHSL     Save SELECT pointer on stack.
            JSR GET.SL     Get operand byte.
            PHA            Save it on the stack.
            JSR INC.SL     Increment SELECT pointer so it points
                           to the opcode following the relative
                           branch instruction. (Relative branches
                           are relative to the next opcode.)
            PLA            Restore operand byte to accumulator.
            CMP #0         Is it plus or minus?
```

```
            BPL FORWRD        If plus, it means a forward branch.
                              Since operand byte is minus, we'll be
                              branching backward.
            DEC SELECT+1      Branching backward is like branching
                              forward from a location 256 bytes lower
                              in memory.
FORWRD      CLC               Add operand byte to the address
            ADC SELECT        of the opcode following the
            BCC RELEND        branch instruction.
            INC SELECT+1
RELEND      STA SELECT        Now SELECT points to the address
                              specified by the operand of the relative
                              branch instruction. Let's print it.
            JSR PR.ADR
            JSR POP.SL        Restore SELECT pointer.
            LDX #1            X holds number of bytes in operand.
            LDA #4            A holds number of characters in
                              operand.
            RTS               Return to caller, with SELECT pointer
                              once again pointing to the operand byte
                              of the relative branch instruction.
```

Zero-Page Mode: ZEROPG

To print the operand of an instruction that uses the zero-page addressing mode, we could simply print a 1-byte operand. But I find listings more readable when all zero-page addresses are shown with the leading zeros (eg: "00FE" rather than "FE" to represent address $00FE). Therefore, let's print all zero-page operands with a leading zero. That simply requires us to print two ASCII zeros and then to print the 1-byte operand. This will cause the bytes "85 2A" to be disassembled as:

```
STA 002A   85 2A
```

```
ZEROPG      LDA #0            Print two ASCII zeroes to all
            JSR PR.BYT        currently selected devices.
            JSR ONEBYT        Print the 1-byte operand.
            LDX #1            X holds number of bytes in operand.
            LDA #4            A holds number of characters in
                              operand.
            RTS               Return to caller.
```

Zero-Page Indexed Modes: ZERO.X, ZERO.Y

To print the operand of an instruction that uses the zero-page X or zero-page Y addressing mode, we need only print the zero-page address, a comma, and then an "X" or a "Y." Thus, "B5 6C" will disassemble as:

```
LDA 006C,X   B5 6C
```

and "B6 53" will disassemble as:

```
LDX 0053,Y   B6 53
```

```
ZERO.X      JSR ZEROPG      Print the zero-page address.
            JSR XINDEX      Print a comma and the letter "X."
            LDX #1          X holds number of bytes in operand.
            LDA #6          A holds number of characters in
                            operand.
            RTS             Return to caller.
ZERO.Y      JSR ZEROPG      Print the zero-page address.
            JSR YINDEX      Print a comma and the letter "Y."
            LDX #1          X holds number of bytes in operand.
            LDA #6          A holds number of characters in
                            operand.
            RTS             Return to caller.
```

A Pseudo-Addressing Mode for Embedded Text

Now we have subroutines to disassemble machine code in any of the 6502's thirteen legal addressing modes. But what about text embedded in a machine-language program? We know that our programs already include text strings, where each text string begins with a TEX character ($7F) and ends with an ETX ($FF). The disassembler, however, doesn't know anything about embedded text. If we try to disassemble a machine-language program that includes embedded text, the disassembler will assume that the TEX character, and the text string itself, are 6502 opcodes and operands; because it doesn't know about text, it will misinterpret the text string.

Wouldn't it be nice if the disassembler could recognize the TEX character for what it is, and then print out the text string *as text*, rather than as opcodes and operands? When it has finished printing a text string, the disassembler could then resume treating the bytes following the ETX as conventional 6502 opcodes and operands.

Such behavior is not hard to implement. We need only define a pseudo-addressing mode, called TEXT mode, and say that the TEX character is the only opcode that has the TEXT addressing mode. Then we'll write a special addressing mode subroutine, called TXMODE, to print operands that are in the TEXT mode. TXMODE will print an operand in the TEXT mode by printing the text that follows the TEX character and ends with the first ETX character.

Here's some source code to implement such behavior:

```
TXMODE   PLA               Pop return address
         PLA               to OPERND.
         PLA               Pop return address
         PLA               to DSLINE.
TXLOOP   JSR NEXTSL        Advance past TEX pseudo-opcode.
         BMI TXEXIT        Return if reached EA.
         JSR GET.SL        Get the character.
         CMP #ETX          Is it the end of the text string?
         BEQ TXEXIT        If so, we've finished disassembling this
                           line.
         JSR PR.CHR        If not, print the character.
         CLC               Branch back to get
         BCC TXLOOP        the next character.
TXEXIT   JSR CR.LF         Advance to a new line.
         JSR NEXTSL        Advance to next opcode (if SELECT is
                           less than EA).
         RTS               Return to the caller of DSLINE, with
                           SELECT at the first opcode following
                           the text string.
```

Now that we have the desired addressing mode subroutines, we can make up the table of addressing mode subroutines:

```
SUBS          .WORD ABSLUT
              .WORD ABS.X
              .WORD ABS.Y
              .WORD ACC
              .WORD IMPLID
              .WORD IMMEDT
              .WORD INDRCT
```

```
        .WORD IND.X
        .WORD IND.Y
        .WORD RELATV
        .WORD ZEROPG
        .WORD ZERO.X
        .WORD ZERO.Y
```

Each addressing mode subroutine will return with SELECT pointing at the last byte in the instruction, with A holding the number of characters in the operand field, and with X holding the number of bytes in the operand (0, 1, or 2). Each addressing mode subroutine will return to OPERND, which will finish the line by calling FINISH.

Finishing the Line: FINISH

FINISH must space over to the proper column for field 3, which will hold the address of the opcode. Then it must print the address of the opcode and dump 1, 2 or 3 bytes, as necessary. FINISH will end by advancing the printhead to a new line and by advancing SELECT so that it points to the first byte following the disassembled line (unless it has disassembled through EA, the ending address, in which case it will return with SELECT = EA). FINISH returns PLUS if more bytes must be disassembled before EA is reached; it returns MINUS if it disassembled through EA.

```
FINISH   STA OPCHRS     Save the length of the operand,
         STX OPBYTS     in characters and in bytes.
         DEX            If necessary, decrement the
         BMI SEL.OK     SELECT pointer so it
LOOP.1   JSR DEC.SL     points to the opcode.
         DEX
         BPL LOOP.1
SEL.OK   SEC            Space over to the
         LDA ADRCOL     column for the address field:
         SBC #4         Operand field started in column 4...
         SBC OPCHRS     ... and includes OPCHRS characters.
         TAX            So now we need X spaces.
         JSR SPACES     Send enough spaces to reach address
                        column.
         JSR PR.ADR     Print address of opcode.
LOOP.2   JSR SPACE      Space once.
         JSR DUMPSL     Dump selected byte.
         JSR INC.SL     Select next byte.
```

```
                DEC OPBYTS      Completed last byte in instruction?
                BPL LOOP.2      If not, do next byte.
                JSR DEC.SL      Back up SELECT to last byte in
                                operand.
FINEND          JSR CR.LF       Advance to a new line.
                RTS             Return to caller.
OPBYTS          .BYTE           Number of bytes in operand.
OPCHRS          .BYTE 0         Number of characters in operand.
ADRCOL          .BYTE 16        Starting column for address field.
```

Now we can disassemble a line. So let's write the disassemblers, one for the printer and one for the screen. These routines will have much the same structure as TVDUMP and PRDUMP, which direct hexdumps to the printer or to the screen.

Disassemble to Screen: TV.DIS

```
TV.DIS          LDA DISLNS      Initialize line counter with
                STA LINUM       number of lines to be disassembled.
                LDA #$FF        Set end address to $FFFF,
                STA EA          so NEXTSL will always increment
                STA EA+1        the SELECT pointer.
                JSR TVT.ON      Select TVT as an output device. (Other
                                selected devices will echo the
                                disassembly.)
TVLOOP          JSR DSLINE      Disassemble one line.
                DEC LINUM       Completed last line yet?
                BNE TVLOOP      If not, disassemble next line.
                RTS             If so, return.
DISLNS          .BYTE 5         DISLNS holds number of lines to be
                                disassembled by TV.DIS. To disassem-
                                ble one line, set DISLNS=1.
LINUM           .BYTE 0         This variable keeps track of the number
                                of lines yet to be disassembled.
```

Printing Disassembler: PR.DIS

The printing disassembler (PR.DIS) will announce itself by displaying "PRINTING DISASSEMBLER" on the screen, but not on the printer. It will then let the user set the starting and ending addresses, in the same manner as PRDUMP. When the user has specified the block of memory to be disassembled, the PR.DIS will print a disassembly of the specified block of memory, echoing its output to the screen.

```
PR.DIS      JSR PR.OFF          Deselect printer.
            JSR TVT.ON          Select TVT.
            JSR PRINT:          Display title:
            .BYTE TEX
            .BYTE CR,LF
            .BYTE               'PRINTING DISASSEMBLER'
            .BYTE CR,LF,ETX
            JSR.SETADS          Let user set starting address
                                and end address.
            JSR GOTOSA          Set SELECT = Start address.
            JSR PR.ON           Select the printer.
PRLOOP      JSR DSLINE          Disassemble one line.
            BPL PRLOOP          If it wasn't the last line, disassemble the
                                next one.
            RTS                 Return to caller.
```

With PR.DIS and TV.DIS, you can disassemble any block of memory, directing the disassembly to the screen or to the printer. See Chapter 12 for guidance on mapping these two disassemblers to function keys in the Visible Monitor.

Chapter 10:

A General MOVE Utility

Many computer programs spend a lot of time moving things from one place to another. Such programs should be able to call a move utility for most of this work. A move utility should:

- Be general enough to move anything of any size from any place in memory to anywhere else.
- Not be upset when the origin block overlaps the destination.
- Have entry points with input configurations convenient to different callers.
- Preserve its inputs.
- Be *fast*.

This routine will be called often. A calling program doesn't want to spend all its time here. The cost of that speed is size, because we'll use straight-line, dedicated code to handle each of several special cases, but even so this move code will weigh in at less than 200 bytes. That's less than three percent of the memory available on a system with 8 K bytes of programmable memory.

Input Configurations

Different callers may find different input configurations convenient, so let's provide more than one entry point, each requiring different parameters to be set. The following two subroutine entry points are likely to meet the needs of most callers:

MOV.EA Move a block, defined by its starting address (SA), its ending

	address (EA), and its destination address (DEST).
MOVNUM	Move a block, defined by its starting address, the number of bytes in the block (NUM), and the destination of the block.

MOV.EA will simply be a "front end" for MOVNUM. It will set NUM = ending address − starting address of the source block.

Handling Overlap

There will be no problem with overlap if we always move from the leading edge of the source block — that is, copy *up* beginning with the highest byte to be moved, and copy *down* beginning with the lowest byte to be moved. This way, if a byte in the source block is overwritten it will already have been copied to its destination.

Going Up?

To avoid overlap, MOVNUM must determine whether it's copying up or down. Therefore, before moving anything it must see if the destination address is greater or lesser than the starting address. Then it can branch to MOVE-UP or MOVE-DOWN as appropriate.

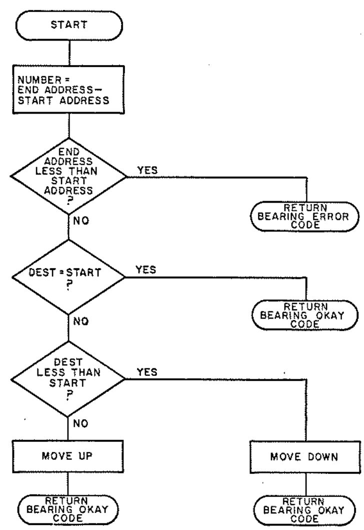

Figure 10.1: *Top level of block move. Flowchart of MOVE.EA and MOV-NUM routines.*

Using the flowchart of figure 10.1 as a guide, let's write source code for the top level of MOV.EA and MOVNUM:

```
        GETPTR = 0          This is the input-page pointer.
        PUTPTR = GETPTR+2   This is the output-page pointer.
MOV.EA  SEC                 Set NUM = EA − SA
        LDX EA+1
        LDA EA
        SBC SA
        STA NUM
        BCS MOVE.1
        DEX
        SEC
MOVE.1  TXA
        SBC SA+1
        STA NUM+1
        BCS MOVNUM          Now NUM = EA − SA.
ER.RTN  LDA #ERROR          If EA less than SA,
        RTS                 return with error code.
MOVNUM  LDY #3              Save the 4 zero-page
SAVE    LDA GETPTR,Y        bytes we'll use.
        PHA
        DEY
        BPL SAVE
        SEC                 Is DEST less than START?
        LDA SA+1
        CMP DEST+1
        BCC MOVEUP          If so, we'll move down.
        BNE MOVEDN          If not, we'll move up.
        LDA SA              SA, destination are in the same
                            page.
        CMP DEST            If SA more than destination, we'll
        BCC MOVEUP          move down. If SA less than destina-
                            tion,
        BNE MOVEDN          we'll move up. If they are equal, we'll
                            return bearing okay code.
OK.RTN  LDY #0              Restore 4 zero-page bytes that were
RESTOR  PLA                 used by the move code.
        STA GETPTR,Y
        INY
        CPY #4              Restored last byte yet?
        BNE RESTOR          If not, restore next one. If so,
        RTS                 return, with move complete and zero
                            page preserved.
NUM     .WORD 0             This 16-bit variable holds the number of
                            bytes to be moved.
```

Optimizing for Speed

Moving a page at a time is the fastest way to move data, and for large blocks we can move most of the bytes this way. Therefore, when moving data we'll move one page at a time until there is less than a page to move; then we'll move a byte at a time until the entire source block is moved. MOVE-UP and MOVE-DOWN must test to see if they have more or less than a page to move, and then branch to dedicated code that either moves a page or moves less than a page.

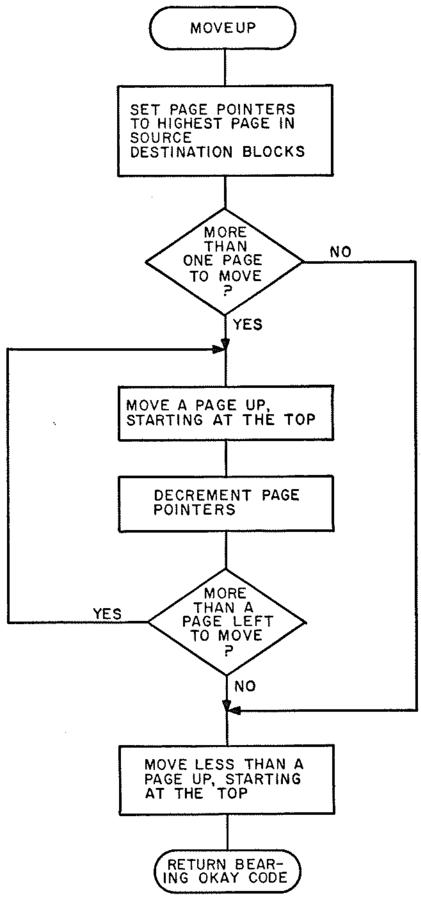

Figure 10.2: *Move a block up. Flowchart of the MOVEUP routine.*

MOVE-UP

Using figure 10.2 as a guide, we can write source code for MOVE-UP:

```
MOVEUP   LDA NUM+1       More than one page to move?
         BEQ LESSUP      If not, move less than a page up.
                         To move more than a page, set the page
                         pointers GETPTR and PUTPTR to the
                         highest pages in the source and destina-
                         tion blocks. To do this, treat X as the
                         high byte and Y as the low byte of a
                         pointer, which we'll call (X,Y). First set
                         (X,Y) = NUM - $FF, the relative ad-
                         dress of the highest page in the block.
         LDY NUM+1       Now Y is high byte of block size.
         LDA NUM         Now A is low byte of block size.
         SEC             Prepare to subtract.
         SBC #$FF.       Now A is a low byte of (block size -
                         $FF.)
         BCS NEXT.1
         DEY
NEXT.1   TAX             Now (X,Y) = NUM - $FF.
                         X is low byte, Y is high byte of NUM -
                         $FF.
         STY PUTPTR+1
         TXA
         CLC             Prepare to add.
         ADC SA
         STA GETPTR
         BCC NEXT.2
         INY
NEXT.2   TYA
         ADC SA+1
         STA GETPTR+1    Now GETPTR = SA + NUM - $FF
                         (the last page in the origin block).
         TXA
         CLC             Prepare to add.
         ADC DEST
         STA PUTPTR
         BCC NEXT.3
         INC PUTPTR+1
NEXT.3   LDA PUTPTR+1
         ADC DEST+1
         STA PUTPTR+1    Now PUTPTR = DEST + NUM - $FF
                         (the last page in the destination block).
                         Now the page pointers (GETPTR and
                         PUTPTR) point to the last page in, respec-
                         tively, the origin and destination blocks.
```

```
         LDX NUM+1         Load X with number of pages to move.
PAGEUP   LDY #$FF          Move a page up.
UPLOOP   LDA (GETPTR),Y    Get a byte from origin block.
         STA (PUTPTR),Y    Put it in destination block.
         DEY               Adjust index for next byte down.
         BNE UPLOOP        Loop if not the last byte.
         LDA (GETPTR),Y    Move last byte.
         STA (PUTPTR),Y
         DEC GETPTR+1      Decrement page pointers.
         DEC PUTPTR+1
         DEX               Still more than a page to move?
         BNE PAGEUP        If so, move up another page.
LESSUP   JSR LOPAGE        Set GETPTR, PUTPTR to bottom of
                           origin and destination blocks.
         LDY NUM           Set index to number of bytes to be
                           moved.
SOMEUP   LDA (GETPTR),Y    Move a byte.
         STA (PUTPTR),Y
         DEY               About to move last byte?
         CPY #$FF
         BNE SOMEUP        If not, move another.
         JMP OK.RTN        If so, return bearing "OK" code.
LOPAGE   LDA SA            Set page pointers to the bottom
         STA GETPTR        of the origin and destination
         LDA SA+1          blocks.
         STA GETPTR+1
         LDA DEST
         STA PUTPTR
         LDA DEST+1
         STA PUTPTR+1
         RTS               Return to caller.
```

Move-Down: MOVEDN

Figure 10.3 shows an algorithm for moving a block of data down through memory.

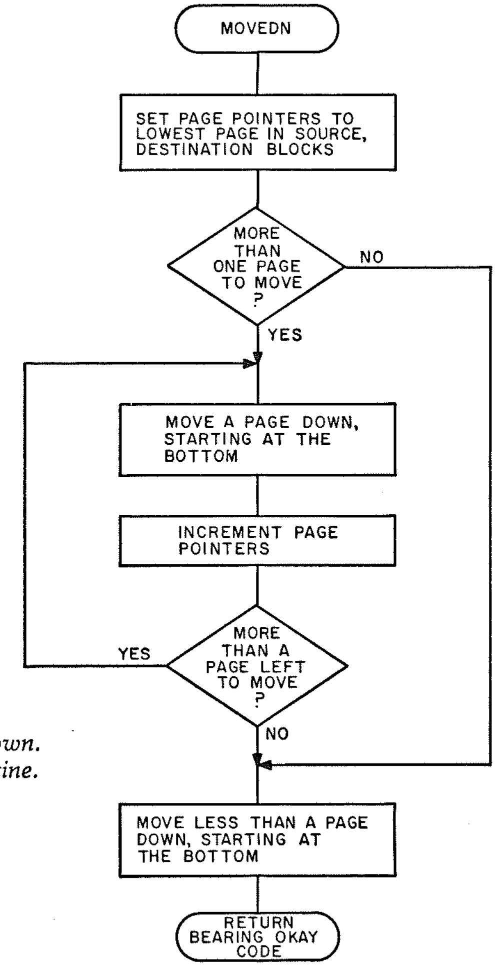

Figure 10.3: *Move a block down. Flowchart of the MOVEDN routine.*

Using figure 10.3 as a guide, we can write source code for the move-down routine:

```
MOVEDN   JSR LOPAGE          Set page pointers to bottom of origin
                             and destination blocks.
         LDY #0              Y must equal zero whether we move
                             more or less than a page.
         LDX NUM+1           More than one page to move?
         BEQ LESSDN          If not, move less than a page down.
                             Move a page down.
PAGEDN   LDA (GETPTR),Y      Get a byte from origin block
         STA (PUTPTR),Y      and put it in destination block.
         INY                 Moved last byte in page?
```

```
        BNE PAGEDN
        INC GETPTR+1      Increment page pointers.
        INC PUTPTR+1
        DEX               Still more than a page to move?
        BNE PAGEDN        If so, move another page down.
        LDY #0            Move less than a page down starting at
                          the bottom.
LESSDN  LDA (GETPTR),Y    Get a byte from origin...
        STA (PUTPTR),Y    and put it in destination block.
        INY               Adjust index for next byte.
        SEC
        CPY NUM           Moved last byte yet?
        BCC LESSDN        If not, move another.
        JMP OK.RTN        If so, return to caller, bearing "OK"
                          code.
```

Speed

For large blocks of data, most bytes will be moved by the page-moving code: PAGE-UP and PAGE-DOWN. Since the processor spends most of its time in these loops, let's see how long they will take to move a byte. (Appendix A5, *Instruction Execution Times*, provides information on the number of cycles required for each 6502 operation.) Ordinarily I would not go into great detail concerning the speed of execution of a small block of code, but these two loops form the heart of the move utility, because they move most of the bytes in any large block. By making those two loops very efficient, we can make the move utility very fast. In fact, these loops will let us move blocks bigger than one page, at a rate approaching 16 cycles/byte moved. (By way of a benchmark, that's more than twice as fast as the time required to move large blocks with MOVIT, a smaller move program published in *The First Book of KIM.** MOVIT, made tiny [95 bytes] to use as little as possible of the KIM's limited programmable memory, requires at least 33 cycles/bytes moved.)

MOVE.EA and MOVNUM are move utilities because they have input configurations and performance suitable for many calling programs. But they are not very convenient to the human user who simply wants to move something. With the Visible Monitor and the move utility, you can move something from one place to

*Butterfield, et al, *The First Book of Kim*, Rochelle Park, NJ: Hayden Book Company, 1977.

another, but you have to know what addresses to set and you have to know the address of the move utility itself.

That's too much for me to remember. I want a *tool,* which will know the addresses and won't require me to remember them.

When I'm developing programs with the Visible Monitor and I want to move some data or code from one place to another, I'd like to be able to call up a move tool with a single keystroke — say "M." It's easier for me to remember " 'M' for Move" than it is to remember the address of the move utility and the addresses of its inputs.

Let's say I'm using the Visible Monitor and I press "M." This invokes the move tool. The first thing it should do is let me know that it's active. What if I hit the "M" key by mistake? The computer should let me know that I've invoked a new program.

It should put up a title: "MOVE TOOL." Then it should let me specify the start, end, and destination addresses of a given block in memory. When these addresses are set, the move tool can call MOV.EA, which will actually perform the move, based on the addresses set by the user.

The top level of the move tool is therefore quite simple. Figure 10.4 shows the flowchart for the following routine:

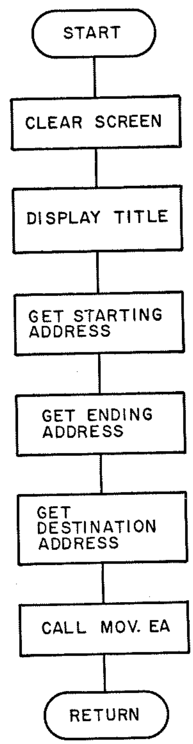

Figure 10.4: *A move tool. Flowchart of MOVER routine.*

MOVER

```
MOVER      JSR TVT.ON           Select screen as an output device.
           JSR PRINT:           Put a title on the screen.
           .BYTE TEX,CR
           .BYTE '  MOVE TOOL'
           .BYTE CR,LF,LF
           .BYTE ETX
           JSR SETADS           Get starting address,
                                ending address, and
           JSR SET.DA           destination address from user.
           JSR MOV.EA           Move the block specified by those
                                pointers.
           RTS                  Return to caller, with requested block
                                moved and with zero page preserved.
```

Of course, MOVER can work only if we have a routine that lets the user set the destination address. Let's write such a routine, and we'll be all set to move whatever we like, to wherever we want it.

Set Destination Address: SET.DA

```
SET.DA     JSR TVT.ON           Select TVT as an output device. All
                                other selected output devices will echo
                                the screen output.
           JSR PRINT:           Put prompt on the screen:
           .BYTE TEX
           .BYTE CR,LF,LF
           .BYTE                "SET DESTINATION ADDRESS  "
           .BYTE                "AND PRESS Q."
           .BYTE ETX
           JSR VISMON           Call the Visible Monitor, so user can
                                specify a given address.
DAHERE     LDA SELECT           Set destination address equal to
           STA DEST             address set by the user.
           LDA SELECT+1
           STA DEST+1
           RTS                  Return to caller.
DEST       .WORD 0              Pointer to destination of block to be
                                moved.
```

See Chapter 12, *Extending the Visible Monitor,* to learn how to hook the move tool into the Visible Monitor by mapping it to a given key. Then to move anything in memory to anywhere else, you need only strike that key and the move tool will do the rest.

Chapter 11:

A Simple Text Editor

With the Visible Monitor you can enter ASCII text into memory by placing the arrow under field 2 and striking character keys. But you must strike two keys for every character in the message: first the character key, to enter the character into the displayed address, and then the space bar, to select the next address. Furthermore, if you want to enter an ASCII space or carriage return into memory, you'll have to place an arrow under field 1 and enter the hexadecimal representation of the desired character: $20 for a space; $0D for a carriage return. Then, of course, you'll have to hit the space bar to select the next address, and the "greater than" key to move the arrow back underneath field 2, so that you can enter the next character into memory.

If you only need to enter up to a dozen ASCII characters at a time, then the Visible Monitor should meet your needs. When you need to enter longer messages into memory, you'll find yourself wanting a more suitable tool — a simple text editor.

Text editors come in many different shapes, sizes and formats. A line-oriented editor, suitable for creating and editing program source files, requires that you enter and edit text a line at a time. Usually each line must be numbered when it is entered; then, in order to edit a line, you must first specify it by its line number.

On the other hand, a character-oriented editor allows you to overstrike, insert, or delete characters anywhere in a given string of characters. Character-oriented editors are frequently found in word processors for office applications, but don't get your hopes up; this chapter will not present software nearly as sophisticated as that available in even the humblest of word processors. However, it will present a very simple character-oriented editor that will enable you to enter and edit text strings, such as prompts, anywhere in memory.

Structure

The text editor will have the three-part structure shown in figure 11.1. From this we can write source code for the top level of the text editor:

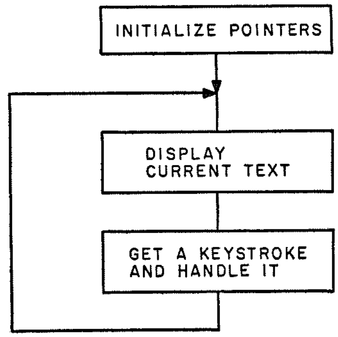

Figure 11.1: *Structure of simple text editor.*

```
EDITOR    JSR SETBUF      Initialize pointers and variables required
                          by the editor.
EDLOOP    JSR SHOWIT      Show the user a portion of the text
                          buffer.
          JSR EDITIT      Let the user edit the buffer or move
                          about within it.
          CLC
          BCC EDLOOP      Loop back to show the current text.
```

Look familiar? It should. This is essentially the same structure used in the Visible Monitor. It's a simple structure, well-suited to the needs of many interactive display programs.

SETBUF

The text editor will operate on text in a portion of memory called the *text buffer*. Because the editor must be able to change the contents of the text buffer, the buffer must occupy programmable memory and may not be used for any other purpose. This exemplifies a problem familiar to programmers: how to allocate memory in the most effective manner. Memory used to store a program cannot be used at the same time to store text; nor can memory allotted to the text buffer be used for storing programs or variables.

How do you get five pounds of tomatoes into a four-pound-capacity sack — without crushing the tomatoes or tearing the sack? You don't. If you want to store a lot of text in your computer's programmable memory, you might not have room for much of a text editor. On the other hand, an elaborate text editor, requiring a good deal of programmable memory for its own code, may not leave much room in your system for storing text.

Therefore, this text editor leaves the allocation of memory for the text buffer to the discretion of the user. A subroutine called SETBUF sets pointers to the starting and ending addresses of the text buffer. The rest of the editor then operates on the text buffer defined by those pointers.

SETBUF sets the starting and ending addresses of the edit buffer. If you always want to enter and edit text in the same buffer, then substitute your own subroutine to set the starting and ending addresses to the values you desire. Otherwise, use the following version of SETBUF, which lets the user define a new text buffer each time it is called.

For testing purposes, you might even want to set the text buffer completely inside screen memory. This allows you to *see* exactly what's happening inside the text buffer.

SETBUF

```
SETBUF    JSR TVT.ON                Select TVT.
          JSR PRINT:                Display "SET UP EDIT BUFFER."
          .BYTE TEX,CR,LF,LF
          .BYTE 'SET UP EDIT BUFFER'
          .BYTE CR,LF,LF,ETX
GETADS    JSR SETADS                Let user set starting address and end ad-
                                    dress of edit buffer.
          JSR GOTOSA                Now SELECT = starting address of edit
                                    buffer.
          RTS                       Return to caller.
```

This version of SETBUF allows the user to set the text buffer anywhere in memory, provided that the ending address is not lower in memory than the starting address. It returns with the SELECT pointer pointing at the starting address of the buffer.

SHOWIT

Now that SETBUF has set the pointers associated with the text buffer, let's figure out how to display part of that buffer.

Figure 11.2 shows the simple 3-line display to be used by the text editor. "X" marks the home position of the edit display. Everything in the edit display is relative to the home position. Thus, to move the edit display about on your screen (ie: from the top of the screen to the bottom of the screen), you need only change the home position, which is set by SHOWIT.

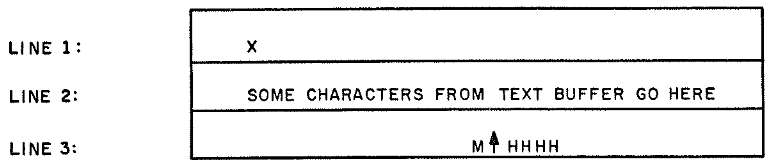

Figure 11.2: *Three-line display of simple text editor.*

Line 1 is entirely blank. Its only purpose is to separate the text displayed in line 2 from whatever you may have above it on your screen.

Line 2 displays a string of characters from the edit buffer. The central character in line 2 is the *current character*. The current character is indicated by an upward-pointing arrow as in line 3. The address of the current character is given by the four hexadecimal characters represented by "HHHH" in line 3.

The letter "M" in line 3 shows you where a graphic character will indicate the current mode of the editor.

Modes

This editor will have two modes: *overstrike mode* and *insert mode*. In overstrike mode you overstrike, or replace, the current character with the character from the keyboard. In insert mode, you insert the keyboard character into the text buffer just before the current character. How one sets these modes, a function for the subroutine EDITIT, will be discussed later. But SHOWIT must know the current mode in order to display the proper graphic in line 3 of the editor display.

Since we're going to have two modes, let's keep track of the current mode of the editor with a 1-byte variable called EDMODE. We'll assign the following values to EDMODE:

EDMODE = 0 when the editor is in overstrike mode.
EDMODE = 1 when the editor is in insert mode.

Any other value of EDMODE is undefined and therefore illegal. If SHOWIT should find that EDMODE has an illegal value, then it should set EDMODE to some legal default value — say, zero. That would make overstrike the default mode for the editor.

We'll also need two graphics characters, INSCHR and OVRCHR, to indicate insert and overstrike modes, respectively. In this chapter, the character to indicate a given edit mode will simply be the first initial of the mode name: "O" for overstrike mode, "I" for insert mode.

SHOWIT

```
SHOWIT  JSR TVPUSH      Save the zero-page bytes we'll use.
        JSR TVHOME      Set home position of the
                        edit display.
        LDX TVCOLS      Clear 3 rows for the
        LDY #3          edit display.
        JSR CLR.XY
        JSR TVHOME      Restore TV.PTR to home position of
                        edit display.
        JSR TVDOWN      Set TV.PTR to beginning of
        JSR TVPUSH      line 2 and save it.
        JSR LINE.2      Display text in line 2.
        JSR TV.POP      Set TV.PTR to beginning
        JSR TVDOWN      of line 3.
        JSR LINE.3      Display line 3.
        JSR TV.POP      Restore zero-page bytes used.
        RTS             Return to caller, with edit display on
                        screen, rest of screen unchanged, and
                        zero page preserved.
```

Of course, SHOWIT can work only if it can call a couple of routines (LINE.2 and LINE.3) to display lines 2 and 3 of the editor display, respectively. Let's write those routines.

Display Text Line

To display the text line, we simply need to copy a number of characters from the text buffer to the second line of the editor display. Since the screen is TVCOLS wide, we should display TVCOLS number of characters in such a way that the central character in the display is the currently selected character. We can do that if we decrement SELECT by TVCOLS/2 times, and then display TVCOLS number of characters:

LINE.2

```
LINE.2    JSR PUSHSL       Save SELECT pointer.
          LDA TVCOLS       Set X equal
          LSR A            to half the width
          TAX              of the screen.
          DEX
          DEX
LOOP.1    JSR DEC.SL       Decrement SELECT X times.
          DEX
          BPL LOOP.1
          LDA TVCOLS       Initialize COUNTR. (We're
          STA COUNTR       going to display TVCOLS characters.)
LOOP.2    JSR GET.SL       Get a character from buffer.
          JSR TV.PUT       Put it on screen.
          JSR TVSKIP       Go to next screen position.
          JSR INC.SL       Advance to next byte in buffer.
          DEC COUNTR       Done last character in row?
          BPL LOOP.2       If not, do next character.
          JSR POP.SL       Restore SELECT from stack.
          RTS              Return to caller.
```

Display Status Line

Line 3 of the editor display provides status information: identifying the current mode of the editor, pointing at the current character in line 2 of the edit display, and providing the address of the current character.

LINE.3

```
LINE.3    LDA TVCOLS
          LSR A           A = TVCOLS/2
          SBC #2          A = (TVCOLS/2) - 2
          JSR TVPLUS      Now TV.PTR is pointing 2 characters to
                          the left of center of line 3 of the edit
                          display.
          LDA EDMODE      What is current mode?
          CMP #1          Is it insert mode?
          BNE OVMODE      If not, it must be overstrike mode.
          LDA #INSCHR     If so, load A with the insert graphic.
          CLC
          BCC TVMODE
OVMODE    LDA #OVRCHR     Load A with the overstrike graphic.
TVMODE    JSR TV.PUT      Put mode graphic on screen.
          LDA #2
          JSR TVPLUS      Now TVPTR is pointing at the center of
                          line 3 of the edit display.
          LDA ARROW       Display an up-arrow here,
          JSR TV.PUT      pointing up at the current character.
          LDA #2
          JSR TVPLUS      Now TV.PTR is pointing at the position
                          reserved for the address of the current
                          character.
          LDA SELECT+1    Display address of current
          JSR VUBYTE      character.
          LDA SELECT
          JSR VUBYTE
          RTS             Return to caller.
```

We've chosen to define the editor's current character as the character pointed to by SELECT. We've already developed some subroutines that operate on the SELECT pointer and on the currently selected byte, so we won't have to write many new editor utilities; instead, we can use many of the SELECT utilities presented in earlier chapters.

Edit Update

Now we can display the three lines of the edit display. What else must the editor do? Oh, yes: it must let us edit. Here's a reasonably useful, if small, set of editor functions:

- Allow the user to move forward through the message.
- Allow the user to move backward through the message.
- Allow the user to overstrike the current character.
- Allow the user to delete the current character.
- Allow the user to delete the entire message.
- Allow the user to insert a new character at the current character position.
- Allow the user to change modes from insert to overstrike and back again.
- Print the message.
- Allow the user to terminate editing, thus causing the editor to return to its caller.

What keys will perform these functions? I'll leave that up to you by treating the editor function keys as variables and keeping them in a table called EDKEYS (see Appendix C11). To assign a given function to a given key, store the character code generated by that key in the appropriate place in the table:

EDITIT

```
EDITIT    JSR GETKEY      Get a keystroke from the user.
          CMP QUITKY      Is it the "quit" key?
          BNE DO.KEY      If not, do what the key requires.
          PHA             Save the key on the stack. If the user
                          gives us 2 "quit" keys in a row, we
                          should exit the editor. So let's see if
                          another QUITKY follows:
          JSR GETKEY
          CMP QUITKY      Is this key a "quit" key?
          BNE NOTEND      If not, then this is not the end of the
                          edit session, so we'd better handle both
                          of those keys, and in their original
                          order.
                          End the edit session:
ENDEDT    PLA             Pop first "quit" key from stack.
          PLA             Pop from stack the return address to
          PLA             the editor's top level.
          RTS             Return to the editor's caller.
NOTEND    STA TEMPCH      Save the key that followed the "quit"
                          key.
          PLA             Pop first "quit" key from stack.
          JSR DO.KEY      Handle it.
          LDA TEMPCH      Restore to the accumulator the key that
                          followed the "quit" key.
```

```
                                "DO.KEY" does what the key in the ac-
                                cumulator requires:
DO.KEY    CMP MODEKY            Is it the "change mode" key?
          BNE IFNEXT            If not, perform the next test.
          DEC EDMODE            If so, change the editor's mode...
          BPL DO.END
          LDA #1
          STA EDMODE
DO.END    RTS                   and return.
IFNEXT    CMP NEXTKY            Is it the "next" key?
          BNE IFPREV            If not, perform the next test.
          JSR NEXTCH            If so, advance the current position by
                                one character...
          RTS                   and return.
IFPREV    CMP PREVKY            Is it the "previous" key?
          BNE IF.RUB            If not, perform the next test.
          JSR PREVCH            If so, back up the current position by
                                one character...
          RTS                   and return.
IF.RUB    CMP RUBKEY            Is it the "delete" key?
          BNE IF.PRT            If not, perform the next test.
          JSR DELETE            If so, delete the current character...
          RTS                   and return.
IF.PRT    CMP PRTKEY            Is it the "print" key?
          BNE IFFLSH            If not, perform the next test.
          JSR PRTBUF            If so, print the buffer...
          RTS                   and return.
IFFLSH    CMP FLSHKY            Is it the "flush" key?
          BNE CHARKY            If not, perform the next test.
          JSR FLUSH             If so, flush all text in the edit buffer...
          RTS                   and return.
                                OK. It's not an editor function key, so it
                                must be a regular character key. There-
                                fore, if we're in overstrike mode we'll
                                overstrike the current character with the
                                new character, and if we're in insert
                                mode we'll insert the new character at
                                the current character position.
CHARKY    LDX EDMODE            Are we in overstrike mode?
          BEQ STRIKE            If so, overstrike the character.
          JSR INSERT            If not, insert the character...
          RTS                   and return.
STRIKE    JSR PUT.SL            Put the character into the currently
                                selected address, which is the address of
```

```
                                 the current character.
          JSR NEXTSL             Advance to the next character position,
          RTS                    and return to caller.
INSERT    PHA                    Save the character to be inserted, while
                                 we make space for it in the edit buffer...
          JSR PUSHSL             Push the address of the current character
                                 onto the stack.
          LDA SA+1               Push starting address of the buffer
          PHA                    onto stack.
          LDA SA
          PHA
          LDA EA+1               Push ending address of the buffer
          PHA                    onto stack.
          LDA EA
          PHA
          JSR SAHERE             Set SA = SELECT, so current character
                                 will be the start of the block we'll move.
          JSR NEXTSL             Advance to next character position in
                                 the text buffer.
          BMI ENDINS             If we're at the end of the buffer, we'll
                                 overstrike instead of inserting.
          JSR DAHERE             Set DEST = SELECT, so destination of
                                 block move will be 1 byte above block's
                                 start address (ie, we'll move a block up
                                 by 1 byte).
          LDA EA                 Decrement end address
          BNE NEXT               so we won't move text
          DEC EA+1               beyond the end of
NEXT      DEC EA                 the text buffer.
                                 Now the starting address is the current
                                 character, the destination address is the
                                 next character, and the ending address is
                                 one character shy of the last character in
                                 the buffer. We're ready now to move a
                                 block.
OPENUP    JSR MOV.EA             Open up 1 byte of space at the current
                                 character's location, by moving to DEST
                                 the block specified by SA and EA.
ENDINS    PLA                    Restore EA so it points to the last byte
          STA EA                 in the edit buffer.
          PLA
          STA EA+1
          PLA                    Restore SA so it points to the first byte
          STA SA                 in the edit buffer.
```

```
            PLA
            STA SA+1
            JSR POP.SL        Restore SELECT so it points to the cur-
                              rent character.
            PLA               Reload the accumulator with the
                              character to be inserted. Since we've
                              created a 1-byte space for this character,
                              we need only overstrike it.
            JSR STRIKE
            RTS               Return to caller.
```

EDITIT looks like it will do what we want it to do — provided that it may call the following (as yet unwritten) subroutines:

- NEXTCH — Select next character.
- PREVCH— Select previous character.
- FLUSH — Flush the buffer.
- PRTBUF — Print the buffer.

Let's write them.

Select Next Character

We want to be able to advance through the text buffer, but we don't want to be able to go beyond the end of the buffer or beyond the end of the message. The end of the message will be indicated by one or more ETX (end-of-text) characters. ETX characters will fill from the last character in the message to the end of the buffer. So if the current character is an ETX, we shouldn't be allowed to advance through memory. Or, if the current character is the last byte in the edit buffer, we shouldn't be allowed to advance through memory. But if we aren't at the end of our text for one reason or another, select the next character by calling the NEXTSL subroutine:

NEXTCH

```
NEXTCH      JSR GET.SL        Get currently selected character.
            CMP #ETX          Is it an ETX?
            BEQ AN.ETX        If so, return to caller, bearing a negative
                              return code.
```

```
         JSR NEXTSL      If not, select next byte in the buffer, and
         RTS             return positive if we incremented
                         SELECT; negative if SELECT already
                         equaled EA.
AN.ETX   LDA #$FF        Since we are on an ETX, we won't incre-
                         ment
         RTS             SELECT; we'll just return with a
                         negative return code.
```

Select Previous Character

The PREVCH (select-previous-character routine) should work in a manner similar to that used by NEXTCH. NEXTCH increments the SELECT pointer and returns *plus*, unless SELECT is greater than or equal to EA, in which case NEXTCH preserves SELECT and returns *minus*. Conversely, PREVCH should decrement SELECT and return *plus*, unless SELECT is less than or equal to SA, in which case it should preserve SELECT and return *minus*:

PREVCH

```
PREVCH   SEC             Prepare to compare.
         LDA SA+1        Is SELECT in a higher page than SA?
         CMP SELECT+1
         BCC SL.OK       If so, SELECT may be decremented.
         BNE NOT.OK      If SELECT is in a lower page than SA,
                         then it's not okay. We'll have to fix it.
                         SELECT is in the same page as SA.
         LDA SA          Is SELECT greater than SA?
         CMP SELECT
         BEQ NO.DEC      If SELECT = SA, don't decrement it.
         BNE NOT.OK      If SELECT is less than SA, it's not okay,
                         so we'll have to fix it.
SL.OK    JSR DEC.SL      SELECT is OK, because it's greater than
                         SA. Thus, we may decrement it and it
                         will remain in the edit buffer.
         LDA #0          Set a positive return code...
         RTS             and return.
NOT.OK   LDA SA          Since SELECT is less than SA, it is
         STA SELECT      not even in the edit buffer. So give
         LDA SA+1        SELECT a legal value, by setting
                         it = SA.
```

```
          STA SELECT+1
          LDA #0           Set a positive return code...
          RTS.             and return.
NO.DEC    LDA #$FF         SELECT = SA, so change nothing. Set
          RTS              a negative return code and return.
```

Flush Buffer

To flush the buffer, we'll just fill the buffer with ETX characters:

FLUSH

```
FLUSH     JSR GOTOSA       Set SELECT to the first character posi-
                           tion in the buffer.
FLOOP     LDA #ETX         Load accumulator with an ETX
                           character...
          JSR PUT.SL       and put it into the buffer.
          JSR NEXTSL       Advance to next byte.
          BPL FLOOP        If we haven't reached the last byte in the
                           buffer, let's repeat the operation for this
                           byte.
          JSR GOTOSA       If we have reached the last byte in the
                           buffer, let's set SELECT to the beginning
                           of the buffer...
          JSR RTS          and return.
```

Print Buffer

To print the buffer, we must print the characters in the edit buffer up to, but not including, the first ETX. Even if there is no ETX in the buffer, we must not print characters from beyond the end of the buffer:

PRTBUF

```
PRTBUF    JSR GOTOSA       Set SELECT to the start of the buffer.
PRLOOP    JSR GET.SL       Get the currently selected character.
          CMP #ETX         Is it an ETX character?
          BEQ ENDPRT       If so, stop printing and return.
```

```
            JSR PR.CHR      If not, print it on all currently selected
                            devices.
            JSR NEXTCH      Advance SELECT by 1 byte within the
                            buffer.
            BPL PRLOOP      If we haven't reached the end of the buf-
                            fer, let's get the next character from the
                            buffer, and handle it.
ENDPRT      RTS             Since we reached the end of the buffer,
                            let's return.
                            When this routine returns, the current
                            character is at the end of the message.
```

Delete Current Character

To delete the current character, we'll take all the characters that follow it in the text buffer and move them to the left by 1 byte. Here's some code to implement such behavior:

```
DELETE      JSR PUSHSL      Save address of current character.
            LDA SA+1        Save buffer's start address.
            PHA
            LDA SA
            PHA
            JSR DAHERE      Set DEST = SELECT, because we'll
                            move a block of text down to here, to
                            close up the buffer at the current
                            character.
            JSR NEXTSL      Advance by 1 byte through text buffer,
                            if possible.
            JSR SAHERE      Set SA = SELECT, because the block
                            we'll move starts 1 byte above the cur-
                            rent character. (Note: the end address of
                            the block we'll move is the end address
                            of the text buffer.)
            JSR MOV.EA      Move block specified by SA, EA, and
                            DEST.
            PLA             Restore initial SA (which
            STA SA          is the start address of the
            PLA             text buffer, not of the block
            STA SA+1        we just moved).
```

JSR POP.SL	Restore SELECT = address of the current character.
RTS	Return to caller.

That's the last of the utilities we need. We now have enough code to comprise a simple text editor. Appendices C10 and C11 are listings of this text editor, showing key assignments that work on an Ohio Scientific C-IP. If you have a different system or prefer your editor functions mapped to different keys, simply change the values of the variables in the key table. If you don't want to have a given function, then for that function store a keycode of zero. You'll find this editor very handy for entering tables of ASCII characters into memory, and for entering, editing, and printing short text strings such as titles for your hexdumps and disassembler listings.

Chapter 12:

Extending the Visible Monitor

At this point you have the Visible Monitor, the print utilities, two hexdump tools, a table-driven disassembler, a move tool, and a simple text editor. Wouldn't it be nice if they were all combined into one interactive software package? Then you could call any tool or function with a single keystroke. Since the Visible Monitor already uses several keys (0 thru 9; A thru F; G; Space; Return; and Rubout or Clear-Screen), we'll have to map these new functions into unused keys.

Here's a list of keys and the functions they will have in the extended monitor:

H	Call a HEXDUMP tool (TVDUMP if the printer is not selected; PRDUMP if the printer is selected).
M	Call MOVER, the move tool.
P	Toggle the printer flag.
T	Call the text editor.
U	Toggle the user output flag.
?	Call the disassembler (TV.DIS if the printer is not selected; PR.DIS if the printer is selected).

With this assignment of keys to functions, we can select or deselect the *printer* at any time just by pressing "P," and likewise the *user*-driven output device just by pressing "U." We can print or display a *hexdump* just by pressing "H" and print or display a disassembly just by pressing "?" (which is almost mnemonic if we think of the disassembler as an answer to our question, "What's in the machine?"). We can move anything from anywhere to anywhere else by pressing "M" for *move*, and we can enter and edit text just by pressing "T" for *text editor.*

Here's some code to provide these features. Since we want to extend the monitor, this subroutine is called EXTEND:

EXTEND

```
                                 When EXTEND is called by the Visible
                                 Monitor's UPDATE routine, a character
                                 from the keyboard is in the ac-
                                 cumulator.
EXTEND    CMP #'P                Is it the "P" key?
          BNE IF.U               If not, perform the next test.
          LDA PRINTR             If so, toggle the
          EOR #$FF               printer flag...
          STA PRINTR
          RTS                    and return to caller.
IF.U      CMP #U                 Is it the "U" key?
          BNE IF.H               If not, perform the next test.
          LDA USR.FN             If so,
          EOR #$FF               toggle the user-output
          STA USR.FN             flag...
          RTS                    and return.
IF.H      CMP #'H                Is it the "H" key?
          BNE IF.M               If not, perform the next test.
          LDA PRINTR             Is the printer selected?
          BNE NEXT.1             If so, print a hexdump.
          JSR TVDUMP             If not, dump to screen...
          RTS                    and return.
NEXT.1    JSR PRDUMP             Print a hexdump...
          RTS                    and return.
IF.M      CMP #'M                Is it the "M" key?
          BNE IF.DIS             If not, perform the next test.
          JSR MOVER              If so, call the move tool.
          RTS                    ...and return.
IF.DIS    CMP #'?                Is it the "?" key?
          BNE IF.T               If not, perform the next test.
          LDA PRINTR             Is the printer selected?
          BNE NEXT.2             If so, print a disassembly.
          JSR TV.DIS             If not, dump to screen...
          RTS                    and return.
NEXT.2    JSR PR.DIS             Print a disassembly...
          RTS                    and return.
IF.T      CMP #'T                Is it the "T" key?
          BNE EXIT               If not, return.
```

```
          JSR EDITOR     If so, call the text editor...
          RTS            and return.
EXIT      RTS            Extend this subroutine by adding more
                         test-and-branch code here.
```

The only remaining step is to modify the Visible Monitor's UPDATE routine so that it calls EXTEND, rather than DUMMY, before it returns. Currently, the Visible Monitor's UPDATE routine calls DUMMY just before it returns, with the bytes $20, $10, and $10 at addresses $13D1, $13D2, and $13D3, respectively. To make the Visible Monitor's UPDATE routine call EXTEND (instead of DUMMY), you must change $13D2 from $10 to $B0.

You can change this byte with the Visible Monitor itself, provided that you are very careful not to touch any key except the keys that are legal to the *un*extended Visible Monitor. Once you have changed $13D2, you may strike any key, but while you are changing $13D2, striking a key that is not legal within the unextended Visible Monitor will cause the Visible Monitor to crash. Be careful. Once you have changed $13D2, try out your new extensions of the Visible Monitor by pressing the now legal keys: "H," "M," "P," "U," "?," and "T."

Chapter 13:
Entering the Software into Your System

Chapters 5 thru 12 present software that will do useful work for you, but only if you can get it into your computer's memory. How you do that will depend on the system you have.

If you have an Apple II, you have an extended machine-language monitor built into your system. If the monitor doesn't come up on RESET, you can invoke it from BASIC with the following BASIC command:

```
POKE 0,0:CALL 0 [RETURN]
```

(The string "[RETURN]" means press the carriage return key.)

This writes a 6502 BRK instruction into location $0000, and then executes a call to a machine-language subroutine at location $0000. The 6502, upon encountering the BRK instruction, will pass control to the Apple II ROM monitor. You'll know you're in the Apple II monitor because you'll see an asterisk (*) on the screen. Your Apple II documentation should tell you how to use this monitor to enter data into memory, dump memory, etc.

The Ohio Scientific C-IP has a much simpler monitor than the Apple II built into its ROM (read-only memory). Press BREAK on the Ohio Scientific C-IP and then press "M." You'll get the ROM monitor display and can use the ROM monitor to enter hexadecimal object code into memory. Unfortunately, although the Ohio Scientific ROM monitor lets you enter a machine-language program into memory by hand, or even from a cassette file in the proper format, it provides no facility for recording a machine-language program onto a cassette. So unless you plan to key the Visible Monitor into memory and then leave your computer on forever, you're out of luck. However, you can SAVE a BASIC program on cassette, and then LOAD it from cassette. And that's the key: we'll use the OSI C-1P's ROM BASIC interpreter to help get machine-language programs into memory.

And what if you have an Atari or a PET Computer? Each of these systems features a BASIC interpreter in ROM (read-only memory), but lacks a machine-language monitor. How can you enter hexadecimal object code into memory using only a BASIC interpreter? Perhaps more importantly, even if we manage to enter that object code into memory, how can we save that object code onto a cassette? If all we have is a BASIC interpreter, the simplest solution is to make our object code look like a BASIC program.

That's not so hard. A BASIC program may contain DATA statements, so a simple BASIC program can contain a number of DATA statements, where the DATA statements actually represent, in decimal, the values of successive bytes in the object code. Then the BASIC program can READ those DATA statements and POKE the values it finds into the appropriate section of memory.

Using BASIC to Load Machine Language

The software in this book can be entered into your computer by RUNning just such a series of BASIC programs. Each of these programs consists of an OBJECT CODE LOADER followed by some number of DATA statements. The first two DATA statements specify the range of DATA statements that follow. Each of the following DATA statements contains ten values: the first value is the start address at which object code from the line is to be loaded; the next eight values represent bytes to be loaded into memory, beginning at the specified address; and the tenth value is the checksum. The checksum is simply the total of the first nine values in the DATA statement. Of these ten values, the first and the tenth will always be greater than 4000, and the others will always be less than 256.

Appendices E1 through E11 contain this book's object code in the form of such DATA statements. You must type each of these DATA statements into your computer, but the BASIC OBJECT CODE LOADER is designed to let you know if you've made a mistake. It won't catch any error you might make while typing, but it will catch the most likely errors. How? The answer is in the checksum. If you make a mistake while typing in one of these DATA lines, the checksum will almost certainly fail to match the sum of the address and the 8 bytes in the line. Then, when the OBJECT CODE LOADER detects a checksum error, it will identify the offending data statement by printing its line number as well as the address specified by the offending line.

The object code loader will use the following variables:

A	The address specified by a data line. Object code from that data line is to be loaded into memory beginning at that address.
BYTE	An array of DIMension 8, containing the values of 8 consecutive bytes of object code as specified by a data line.
CHECK	The checksum specified by a data line.
FIRST	The number of the first DATA statement containing object code.
LAST	The number of the last DATA statement containing object code.
LINE	A line counter, tracking the number of data lines of object code already loaded into memory.
SUM	The calculated sum of the 8 bytes of object code and the address specified by a given data line. If SUM equals the checksum specified by that data line, then the data is probably correct.
TEMP	A temporary variable.

Here is the object code loader:

```
100 REM                           OBJECT CODE LOADER by Ken Skier
110 REM
120 DIM BYTE(8)                   :REM Initialize BYTE array.
130 READ FIRST                    :REM Get the line number of the first
140 REM                           DATA statement containing object code.
150 READ LAST                     :REM Get the line number of the last
160 REM                           DATA statement containing object code.
170 FOR LINE=FIRST TO LAST        :REM Read the specified DATA lines.
180 GOSUB 300                     :REM Load next data line into memory.
190 NEXT LINE                     :REM If not done, read next DATA line.
200 PRINT "LOADED LINES",FIRST,"THROUGH",LAST,"SUCCESSFULLY."
210 END                           :REM If done, say so.
220 REM
230 REM                           Subroutine at 300 handles one
240 REM                           DATA statement.
300 READ A                        :REM Get address for object code.
310 SUM=A                         :REM Initialize calculated sum of data.
320 FOR J=1 TO 8                  :REM Get 8 bytes of object code from
321 REM                           data.
330 READ TEMP: BYTE(J)=TEMP       :REM Put them in the byte array, and
340 SUM=SUM+BYTE(J)               :REM add them to the calculated sum of
341 REM                           data.
350 NEXT J                        :REM Now we have the 8 bytes, and we
360 REM                           have calculated the sum of the data.
370 READ CHECK                    :REM Get checksum from data line.
380 IF SUM <>CHECK THEN 500       :REM If checksum error, handle it.
```

```
390 FOR J=1 TO 8                    :REM Since there is no checksum error,
400 POKE A+J-1,BYTE(J)              :REM poke the data into the specified
410 NEXT J                          :REM portion of memory,
420 RETURN                          :REM and return to caller.
430 REM
440 REM                             Checksum error-handling code follows.
500 PRINT "CHECKSUM ERROR IN DATA LINE",LINE
510 PRINT "START ADDRESS GIVEN IN BAD DATA LINE IS", A
520 END
530 REM                             The next two DATA statements specify
540 REM                             the range of DATA statements that
550 REM                             contain object code.
570 REM
600 DATA ????                       :REM This should be the number of the
610 REM                             first DATA statement containing object
611 REM                             code.
612 REM
620 DATA ????                       :REM This should be the number of the
630 REM                             last DATA statement containing object
631 REM                             code.
```

Once you've entered the BASIC OBJECT CODE LOADER into your computer's memory, SAVE it on a cassette. Remember that by itself the BASIC OBJECT CODE LOADER can do nothing; it needs DATA statements in the proper form to be a complete, useful program. When you're ready to create such a program, LOAD the BASIC OBJECT CODE LOADER from cassette back into memory. Now you're ready to append to it DATA statements from one of the E Appendices — for example, from Appendix E1. Do not append DATA statements from more than one appendix to the same BASIC program. Append as many DATA lines as you can, without using memory above $0FFF (decimal 4095). You can insure that you don't run over this limit by setting 4095 as the top of memory available to your system's BASIC interpreter. How do you set the top of memory available to the BASIC interpreter? That varies from system to system, so consult the B Appendix for your system.

Before you can append to the OBJECT CODE LOADER all the DATA statements from Appendix E1, your BASIC interpreter may give you an OUT OF MEMORY error (MEMORY FULL). When that happens, delete the last DATA line you appended to the OBJECT CODE LOADER. Let's say you've appended DATA lines 1000 thru 1022 when you get an OUT OF MEMORY error. Delete DATA line 1022. Now enter the line numbers of the first and last of the object code DATA statements into DATA lines 600 and 620, like this:

```
600   DATA   1000
620   DATA   1021
```

DATA lines 600 and 620, the very first DATA lines in your program, tell the BASIC OBJECT CODE LOADER how many DATA lines of object code follow. Now the OBJECT CODE LOADER can "know" how many DATA lines to read, without reading too few or too many. In this case, DATA lines 600 and 620 tell the OBJECT CODE LOADER that the object code may be found in DATA lines 1000 thru 1021.

Note that DATA lines 600 and 620 each contain one value, whereas the remaining DATA lines each contain ten values.

Now you are ready to RUN the OBJECT CODE LOADER. Unless you're a better typist than I am, you probably made some mistakes while typing in the DATA lines from Appendix E1. Don't worry; the incorrect data will not be blindly loaded into memory. If the BASIC OBJECT CODE LOADER detects a checksum error, it will tell you so, like this:

```
CHECKSUM ERROR IN DATA STATEMENT              1012
START ADDRESS GIVEN IN BAD DATA LINE IS       4442
```

This means that data statement 1012 has a checksum error: ie, bad data. To help you double check, the second line of the error message specifies the start address given by the bad data line: this is the first number in the offending data line. These two items of information should make it easy for you to find the bad data line—just look for the DATA statement whose line number is 1012 and whose first value is 4442. That's the DATA statement you entered incorrectly. Now you need only eyeball the ten numbers in that line, comparing them to the corresponding DATA statement in Appendix E1, and you should quickly find the number or numbers you entered incorrectly. Fix that DATA statement, and RUN the LOADER again.

When you have entered all of the DATA statements correctly, RUNning the LOADER will load the object code they specify into memory. The OBJECT CODE LOADER will then print:

```
LOADED LINES aaaa THROUGH bbbb SUCCESSFULLY
```

where 'aaaa' is the number of the first DATA line of object code, and 'bbbb' is the number of the last DATA line of object code in the program. This message tells you that the BASIC OBJECT CODE LOADER has read and POKE'd the indicated range of DATA statements into memory.

When you see this message, you have verified the program, so SAVE it on a cassette. Then make up a new BASIC program, containing the OBJECT CODE LOADER and the next group of DATA statements from an E Appendix. (Remember not to append DATA lines from more than one E Appendix to the same BASIC program.) Store in lines 600 and 620 the line numbers of the first and last DATA statements you copied from the E Appendix. Verify and SAVE this program as well, and then continue in this manner until you have entered, verified, and SAVE'd BASIC programs containing all of the DATA statements in Appendices E1 thru E10, as well as the DATA statements in the E Appendix containing system data for your computer (one of the Appendices E11 thru E14). RUNning all of those BASIC programs will then enter all of the software presented in this book into your computer's memory.

At this point, you should be ready to transfer control from your computer's BASIC interpreter to the VISIBLE MONITOR.

Activating the Visible Monitor

Once you have entered the object code for the Screen Utilities, the Visible Monitor, and the System Data Block into your system, you can activate the Visible Monitor by causing the 6502 in your computer to execute a JSR (jump to subroutine) to $1207.

Using the Ohio Scientific C-IP ROM monitor, you can activate the Visible Monitor simply by typing:

```
1207G
```

Using the Apple II ROM monitor, you can call the Visible Monitor with the command:

```
G1207 [RETURN]
```

Using the Atari 400 or 800 with its BASIC cartridge plugged in, you can invoke the Visible Monitor with the BASIC command:

X=USR(4615) [RETURN]

In Atari BASIC, you can call a machine-language subroutine by passing the address of that subroutine as a parameter to the USR function. Since $1207 is 4615 in decimal, the command X=USR(4615) causes Atari BASIC to call the subroutine at $1207. (The value returned by that subroutine will then be stored in the BASIC variable X — not in the 6502's X register. But that doesn't concern us because the Visible Monitor isn't designed to return a value to its caller.)

Using the PET 2001, you can invoke the Visible Monitor from BASIC in the immediate mode with the following BASIC command:

SYS (4615)

When you press (RETURN), you'll see the Visible Monitor display, because SYS (4615) causes BASIC to call the subroutine at address 4615 decimal, which is $1207—the entry point for the Visible Monitor.

If and when you press "Q" to quit the Visible Monitor, the Visible Monitor will return to its caller — PET BASIC. (The Visible Monitor doesn't leave much room for a PET BASIC program, since your BASIC program and its arrays, variables, etc cannot require memory beyond $0FFF, but the Visible Monitor should work very well with a small PET BASIC program. In any case, it's reassuring to have a new program such as the Visible Monitor return to a familiar one such as the PET BASIC interpreter.)

Once you have activated the Visible Monitor, you should see its display on the screen. If you don't see such a display, then the Visible Monitor has not been entered properly into your system's memory; perhaps you failed to enter the display code properly.

If you do see the Visible Monitor display on the screen, press the space bar. The display should change — specifically, the displayed address should increment, and fields 1 and 2, immediately to the right of the displayed address, may also change.

If nothing changes when you press the space bar, then the display code probably works fine, but you failed to enter the UPDATE code properly.

If the space bar does change the display, then test out the other functions of the Visible Monitor: press RETURN to decrement the selected address; press hexadecimal keys to select a different address; then select an address somewhere in screen memory and place new data into that address. If you picked a place in display memory that is not cleared by the Visible Monitor (ie: a place not in the top five rows of the screen), then you should be able to place arbitrary characters on the screen just by using the Visible Monitor to store arbitrary values in the selected address.

If your Visible Monitor fails to perform properly, you may have entered it into memory incorrectly. Compare the DATA statements you appended to the OBJECT

CODE LOADER with the DATA statements in the E Appendices. Remember: if even 1 byte is entered incorrectly, then in all likelihood the Visible Monitor will fail to function.

To extend the Visible Monitor as described in Chapter 12, store a $BO in address $13D2. To disable the features described in Chapter 12, store a $10 in address $13D2. Now you're really getting your hands on the machine, reaching into memory and operating on the bytes, and with that kind of control, you can do almost anything.

NOTE:

The author intends to provide the software in this book for sale on cassettes compatible with the Apple II, Atari, Ohio Scientific, and PET computers. If you prefer to load your software from cassette, rather than enter it in by hand, contact the author through BYTE Books.

Appendices

Appendix A1:

Hexadecimal Conversion Table

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	158	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

Appendix A2:

ASCII Character Codes

Code	Char	Code	Char	Code	Char	Code	Char
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Appendix A3:

6502 Instruction Set — Mnemonic List

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BML	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Register X
CPY	Compare Memory and Register Y
DEC	Decrement Memory
DEX	Decrement Register X
DEY	Decrement Register Y
EOR	"Exclusive Or" Memory with Accumulator
INC	Increment Memory
INX	Increment Register X
INY	Increment Register Y

JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Register X with Memory
LDY	Load Register Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Register X in Memory
STY	Store Register Y in Memory
TAX	Transfer Accumulator to Register X
TAY	Transfer Accumulator to Register Y
TSX	Transfer Stack Pointer to Register X
TXA	Transfer Register X to Accumulator
TXS	Transfer Register X to Stack Pointer
TYA	Transfer Register Y to Accumulator

Appendix A4:

6502 Instruction Set — Opcode List

00 — BRK
01 — ORA — (Indirect,X)
02 — Future Expansion
03 — Future Expansion
04 — Future Expansion
05 — ORA — Zero Page
06 — ASL — Zero Page
07 — Future Expansion
08 — PHP
09 — ORA — Immediate
0A — ASL — Accumulator
0B — Future Expansion
0C — Future Expansion
0D — ORA — Absolute
0E — ASL — Absolute
0F — Future Expansion

10 — BPL
11 — ORA — (Indirect),Y
12 — Future Expansion
13 — Future Expansion
14 — Future Expansion
15 — ORA — Zero Page,X
16 — ASL — Zero Page,X
17 — Future Expansion
18 — CLC
19 — ORA — Absolute,Y
1A — Future Expansion
1B — Future Expansion
1C — Future Expansion
1D — ORA — Absolute, X
1E — Future Expansion
1F — Future Expansion

20 — JSR
21 — AND — (Indirect,X)
22 — Future Expansion
23 — Future Expansion
24 — Bit — Zero Page
25 — AND — Zero Page
26 — ROL — Zero Page
27 — Future Expansion
28 — PLP
29 — AND — Immediate
2A — ROL — Accumulator
2B — Future Expansion
2C — BIT — Absolute
2D — AND — Absolute
2E — ROL — Absolute
2F — Future Expansion

30 — BMI
31 — AND — (Indirect),Y
32 — Future Expansion
33 — Future Expansion
34 — Future Expansion
35 — AND — Zero Page,X
36 — ROL — Zero Page,X
37 — Future Expansion
38 — SEC
39 — AND — Absolute,Y
3A — Future Expansion
3B — Future Expansion
3C — Future Expansion
3D — AND — Absolute,X
3F — Future Expansion

40 — RTI
41 — EOR — (Indirect,X)
42 — Future Expansion
43 — Future Expansion
44 — Future Expansion
45 — EOR — Zero Page
46 — LSR — Zero Page
47 — Future Expansion
48 — PHA
49 — EOR — Immediate
4A — LSR — Accumulator
4B — Future Expansion
4C — JMP — Absolute
4D — EOR — Absolute
4E — LSR — Absolute
4F — Future Expansion

50 — BVC
51 — EOR — (Indirect),Y
52 — Future Expansion
53 — Future Expansion
54 — Future Expansion
55 — EOR — Zero Page,X
56 — Zero Page,X
57 — Future Expansion
58 — CLI
59 — EOR — Absolute,Y
5A — Future Expansion
5B — Future Expansion
5C — Future Expansion
5D — EOR — Absolute,X
5E — LSR — Absolute,X
5F — Future Expansion

60 — RTS
61 — ADC — (Indirect,X)
62 — Future Expansion
63 — Future Expansion
64 — Future Expansion
65 — ADC — Zero Page
66 — ROR — Zero Page
57 — Future Expansion
68 — PLA
69 — ADC — Immediate
6A — ROR — Accumulator
6B — Future Expansion
6C — JMP — Indirect
6D — ADC — Absolute
6E — ROR — Absolute
6F — Future Expansion

70 — BVS
71 — ADC — (Indirect),Y
72 — Future Expansion
73 — Future Expansion
74 — Future Expansion
75 — ADC — Zero Page,X
76 — ROR — Zero Page,X
77 — Future Expansion
78 — SEI
79 — ADC Absolute,Y
7A — Future Expansion
7B — Future Expansion
7C — Future Expansion
7D — ADC — Absolute,X
7E — ROR — Absolute,X
7F — Future Expansion

80 — Future Expansion
81 — STA — (Indirect,X)
82 — Future Expansion
83 — Future Expansion
84 — STY — Zero Page
85 — STA — Zero Page
86 — STX — Zero Page
87 — Future Expansion
88 — DEY
89 — Future Expansion
8A — TXA
8B — Future Expansion
8C — STY — Absolute
8D — STA — Absolute
8E — STX — Absolute
8F — Future Expansion

90 — BCC
91 — STA — (Indirect),Y
92 — Future Expansion
93 — Future Expansion
94 — STY — Zero Page,X
95 — STA — Zero Page,X
96 — STX — Zero Page,Y
97 — Future Expansion
98 — TYA
99 — STA — Absolute,Y
9A — TXS
9B — Future Expansion
9C — Future Expansion
9D — STA — Absolute,X
9E — Future Expansion
9F — Future Expansion

A0 — LDY — Immediate
A1 — LDA — (Indirect,X)
A2 — LDX — Immediate
A3 — Future Expansion
A4 — LDY — Zero Page
A5 — LDA — Zero Page
A6 — LDX — Zero Page
A7 — Future Expansion
A8 — TAY
A9 — LDA — Immediate
AA — TAX
AB — Future Expansion
AC — LDY — Absolute
AD — LDA — Absolute
AE — LDX — Absolute
AF — Future Expansion

B0 — BCS
B1 — LDA — (Indirect),Y
B2 — Future Expansion
B3 — Future Expansion
B4 — LDY — Zero Page,X
B5 — LDA — Zero Page,X
B6 — LDX — Zero Page,Y
B7 — Future Expansion
B8 — CLV
B9 — LDA — Absolute,Y
BA — TSX
BB — Future Expansion
BC — LDY — Absolute,X
BD — LDA — Absolute,X
BE — LDX — Absolute,Y
BF — Future Expansion

C0 — CPY — Immediate
C1 — CMP — (Indirect,X)
C2 — Future Expansion
C3 — Future Expansion
C4 — CPY — Zero Page
C5 — CMP — Zero Page
C6 — DEC — Zero Page
C7 — Future Expansion
C8 — INY
C9 — CMP — Immediate
CA — DEX
CB — Future Expansion
CC — CPY — Absolute
CD — CMP — Absolute
CE — DEC — Absolute
CF — Future Expansion

D0 — BNE
D1 — CMP — (Indirect),Y
D2 — Future Expansion
D3 — Future Expansion
D4 — Future Expansion
D5 — CMP — Zero Page,X
D6 — DEC — Zero Page,X
D7 — Future Expansion
D8 — CLD
D9 — CMP — Absolute,Y
DA — Future Expansion
DB — Future Expansion
DC — Future Expansion
DD — CMP — Absolute,X
DE — DEC — Absolute,X
DF — Future Expansion

E0 — CPX — Immediate
E1 — SEC — (Indirect,X)
E2 — Future Expansion
E3 — Future Expansion
E4 — CPX — Zero Page
E5 — SBC — Zero Page
E6 — Zero Page
E7 — Future Expansion
E8 — INX
E9 — SBC — Immediate
EA — NOP
EB — Future Expansion
EC — CPX — Absolute
ED — SBC — Absolute
EE — INC — Absolute
EF — Future Expansion

F0 — BEQ
F1 — SBC — (Indirect),Y
F2 — Future Expansion
F3 — Future Expansion
F4 — Future Expansion
F5 — SBC — Zero Page,X
F6 — INC — Zero Page,X
F7 — Future Expansion
F8 — SED
F9 — SBC — Absolute,Y
FA — Future Expansion
FB — Future Expansion
FC — Future Expansion
FD — SBC — Absolute,X
FE — INC — Absolute,X
FF — Future Expansion

Appendix A5:

Instruction Execution Times (in clock cycles)

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect), X	(Indirect), Y	Absolute Indirect
ADC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
AND	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
ASL	2	.	5	6	.	6	7	.	.	.	.	.	.
BCC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BCS	.	.	.	.	.	.	.	.	.	2**	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	2**	.	.	.
BIT	.	.	3	.	.	4	.	.	.	.	.	.	.
BMI	.	.	.	.	.	.	.	.	.	2**	.	.	.
BNE	.	.	.	.	.	.	.	.	.	2**	.	.	.
BPL	.	.	.	.	.	.	.	.	.	2**	.	.	.
BRK	.	.	.	.	.	.	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	2**	.	.	.
BVS	.	.	.	.	.	.	.	.	.	2**	.	.	.
CLC	.	.	.	.	.	.	.	.	2	.	.	.	.
CLD	.	.	.	.	.	.	.	.	2	.	.	.	.
CLI	.	.	.	.	.	.	.	.	2	.	.	.	.
CLV	.	.	.	.	.	.	.	.	2	.	.	.	.
CMP	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
CPX	.	2	3	.	.	4	.	.	.	.	.	.	.
CPY	.	2	3	.	.	4	.	.	.	.	.	.	.
DEC	.	.	5	6	.	6	7	.	.	.	.	.	.
DEX	.	.	.	.	.	.	.	.	2	.	.	.	.
DEY	.	.	.	.	.	.	.	.	2	.	.	.	.
EOR	.	2	3	4	.	4	4*	4*	.	.	6	5	.

	Accumulator	Immediate	Zero Page	Zero Page, X	Zero Page, Y	Absolute	Absolute, X	Absolute, Y	Implied	Relative	(Indirect), X	(Indirect), Y	Absolute Indirect
INC	.	.	5	6	.	6	7	.	.	.	.	.	.
INX	.	.	.	.	.	.	.	.	2	.	.	.	.
INY	.	.	.	.	.	.	.	.	2	.	.	.	.
JMP	.	.	.	.	.	3	.	.	.	.	.	.	5
JSR	.	.	.	.	.	6	.	.	.	.	.	.	.
LDA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
LDX	.	2	3	.	4	4	.	4*	.	.	.	.	.
LDY	.	2	3	4	.	4	4*	.	.	.	.	.	.
LSR	2	.	5	6	.	6	7	.	.	.	.	.	.
NOP	.	.	.	.	.	.	.	.	2	.	.	.	.
ORA	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
PHA	.	.	.	.	.	.	.	.	3	.	.	.	.
PHP	.	.	.	.	.	.	.	.	3	.	.	.	.
PLA	.	.	.	.	.	.	.	.	4	.	.	.	.
PLP	.	.	.	.	.	.	.	.	4	.	.	.	.
ROL	2	.	5	6	.	6	7	.	.	.	.	.	.
ROR	2	.	5	6	.	6	7	.	.	.	.	.	.
RTI	.	.	.	.	.	.	.	.	6	.	.	.	.
RTS	.	.	.	.	.	.	.	.	6	.	.	.	.
SBC	.	2	3	4	.	4	4*	4*	.	.	6	5*	.
SEC	.	.	.	.	.	.	.	.	2	.	.	.	.
SED	.	.	.	.	.	.	.	.	2	.	.	.	.
SEI	.	.	.	.	.	.	.	.	.	.	.	.	.
STA	.	.	3	4	.	4	5	5	.	.	6	6	.
STX*	.	.	3	.	4	4	.	.	.	.	.	.	.
STY**	.	.	3	4	.	4	.	.	.	.	.	.	.
TAX	.	.	.	.	.	.	.	.	2	.	.	.	.
TAY	.	.	.	.	.	.	.	.	2	.	.	.	.
TSX	.	.	.	.	.	.	.	.	2	.	.	.	.
TXA	.	.	.	.	.	.	.	.	2	.	.	.	.
TXS	.	.	.	.	.	.	.	.	2	.	.	.	.
TYA	.	.	.	.	.	.	.	.	2	.	.	.	.

* Add one cycle if indexing across page boundary

** Add one cycle if branch is taken, Add one additional if branching operation crosses page boundary

Appendix A6:

6502 Opcodes by Mnemonic and Addressing Mode

Addressing Modes

Mnemonics	ABSOLUTE	ABSOLUTE,X	ABSOLUTE,Y	ACCUMULATOR	IMMEDIATE	IMPLIED	INDIRECT	INDIRECT,X	INDIRECT,Y	RELATIVE	ZERO PAGE	ZERO PAGE,X	ZERO PAGE,Y
ADC	6D	7D	79	.	69	.	.	61	71	.	65	75	.
AND	2D	3D	39	.	29	.	.	21	31	.	25	35	.
ASL	0E	1E	.	0A	.	.	.	.	.	.	06	16	.
BCC	.	.	.	.	.	.	.	.	.	90	.	.	.
BCS	.	.	.	.	.	.	.	.	.	B0	.	.	.
BEQ	.	.	.	.	.	.	.	.	.	F0	.	.	.
BIT	2C	.	.	.	.	.	.	.	.	.	24	.	.
BMI	.	.	.	.	.	.	.	.	.	30	.	.	.
BNE	.	.	.	.	.	.	.	.	.	D0	.	.	.
BPL	.	.	.	.	.	.	.	.	.	10	.	.	.
BRK	.	.	.	.	.	00	.	.	.	.	.	.	.
BVC	.	.	.	.	.	.	.	.	.	50	.	.	.
BVS	.	.	.	.	.	.	.	.	.	70	.	.	.
CLC	.	.	.	.	.	18	.	.	.	.	.	.	.
CLD	.	.	.	.	.	D8	.	.	.	.	.	.	.
CLI	.	.	.	.	.	58	.	.	.	.	.	.	.

	Addressing Modes												
Mnemonics	ABSOLUTE	ABSOLUTE,X	ABSOLUTE,Y	ACCUMULATOR	IMMEDIATE	IMPLIED	INDIRECT	INDIRECT,X	INDIRECT,Y	RELATIVE	ZERO PAGE	ZERO PAGE,X	ZERO PAGE,Y
CLV	.	.	.	.	.	B8	.	.	.	.	.	.	.
CMP	CD	DD	D9	.	C9	.	.	C1	D1	.	C5	D5	.
CPX	EC	.	.	.	E0	.	.	.	.	.	E4	.	.
CPY	CC	.	.	.	C0	.	.	.	.	.	C4	.	.
DEC	CE	DE	.	.	.	.	.	.	.	.	C6	D6	.
DEX	.	.	.	.	.	CA	.	.	.	.	.	.	.
DEY	.	.	.	.	.	88	.	.	.	.	.	.	.
EOR	4D	5D	59	.	49	.	.	41	51	.	45	55	.
INC	EE	FE	.	.	.	.	.	.	.	.	E6	F6	.
INX	.	.	.	.	.	E8	.	.	.	.	.	.	.
INY	.	.	.	.	.	C8	.	.	.	.	.	.	.
JMP	4C	.	.	.	.	.	6C	.	.	.	.	.	.
JSR	20	.	.	.	.	.	.	.	.	.	.	.	.
LDA	AD	BD	B9	.	A9	.	.	A1	B1	.	A5	B5	.
LDX	AE	.	BE	.	A2	.	.	.	.	.	A6	.	.
LDY	AC	BC	.	.	A0	.	.	.	.	.	A4	B4	.
LSR	4E	5E	.	4A	.	.	.	.	.	.	46	56	.
NOP	.	.	.	.	.	EA	.	.	.	.	.	.	.
ORA	0D	1D	19	.	09	.	.	01	11	.	05	15	.
PHA	.	.	.	.	.	48	.	.	.	.	.	.	.
PHP	.	.	.	.	.	08	.	.	.	.	.	.	.
PLA	.	.	.	.	.	68	.	.	.	.	.	.	.
PLP	.	.	.	.	.	28	.	.	.	.	.	.	.
ROL	2E	3E	.	2A	.	.	.	.	.	.	26	36	.
ROR	6E	7E	.	6A	.	.	.	.	.	.	66	76	.
RTI	.	.	.	.	.	40	.	.	.	.	.	.	.

Mnemonics	ABSOLUTE	ABSOLUTE,X	ABSOLUTE,Y	ACCUMULATOR	IMMEDIATE	IMPLIED	INDIRECT	INDIRECT,X	INDIRECT,Y	RELATIVE	ZERO PAGE	ZERO PAGE,X	ZERO PAGE,Y
RTS	.	.	.	.	.	60	.	.	.	.	.	.	.
SBC	ED	FD	F9	.	E9	.	.	E1	F1	.	E5	F5	.
SEC	.	.	.	.	.	38	.	.	.	.	.	.	.
SED	.	.	.	.	.	F8	.	.	.	.	.	.	.
SEI	.	.	.	.	.	78	.	.	.	.	.	.	.
STA	8D	9D	99	.	.	.	.	81	91	.	85	95	.
STX	8E	.	.	.	.	.	.	.	.	.	86	.	.
STY	8C	.	.	.	.	.	.	.	.	.	84	94	.
TAX	.	.	.	.	.	AA	.	.	.	.	.	.	.
TAY	.	.	.	.	.	A8	.	.	.	.	.	.	.
TSX	.	.	.	.	.	BA	.	.	.	.	.	.	.
TXA	.	.	.	.	.	8A	.	.	.	.	.	.	.
TXS	.	.	.	.	.	9A	.	.	.	.	.	.	.
TYA	.	.	.	.	.	98	.	.	.	.	.	.	.

Appendix B1:

The Ohio Scientific Challenger I-P

The Ohio Scientific Challenger I-P is the simplest of the systems considered in this book. Its screen is mapped in the manner described in Chapter 5: the lowest screen address is in the upper left corner, and the screen addresses increase uniformly as you move to the right and down the screen. Any ASCII character stored in screen memory will be displayed properly on the video screen; it is not necessary to replace the ASCII character with a system-specific display code. Therefore, the system data block may be initialized as shown in Appendices C13 and E12.

Incidentally, the OSI C-IP's screen TVT subroutine at $BF2D stores the relative location of the cursor in $0200. Modify $0200 and you change the next location at which a character will be printed to the screen.

If you have an Ohio Scientific BASIC-in-ROM system other than the Challenger I-P, it may have different character input/output routines. If so, examine the following locations:

BASIN	$FFEB	General character-input routine for OSI BASIC-in-ROM.
BASOUT	$FFEE	General character-output routine for OSI BASIC-in-ROM.

For example, in the OSI C-IP you can get a character from the keyboard by calling $FEED, or you may call OSI's general character-input routine at $FFEB. This routine gets a character from the keyboard unless the SAVE flag is set, in which case it gets a character from the cassette input port. Similarly, in the OSI C-IP you can print a character to the screen by calling $BF2D, or send a character to the cassette output port by calling $FCB1. Or, you can simply call OSI's general character-output routine at $FFEE, which outputs the accumulator to the screen and, if the SAVE flag is set, echoes to the serial port as well.

Thus, even if you don't know the addresses of your OSI system's specific I/O routines, you can set ROMKEY=$FFEB and ROMTVT=$FFEE. When you RESET your system, the Ohio Scientific Operating System will automatically "hook" those routines to your keyboard for input and to your screen for output.

Setting the Top of Memory

If you wish to load object code using the BASIC OBJECT CODE LOADER (see Chapter 13) you must first set the top of memory available to your BASIC interpreter to $0FFF. Do this as part of cold-starting BASIC. To cold-start BASIC, turn on your OSI computer, press the (BREAK) key, and then press 'C'. The screen will prompt, "Memory Size?" Type "4095" and then press (RETURN). Now BASIC will use the lowest 4K of RAM, leaving memory from $1000 and up available to machine-language programs.

With the top of memory set to $0FFF, you may enter and RUN the BASIC programs that load object code into your computer's memory.

Calling Machine-Language Code from BASIC

To call a machine-language subroutine from BASIC, first set the pointer at $000B, 000C so it points to the subroutine, and then call that subroutine with BASIC's USR function, either in the immediate mode or from within a BASIC program. For example, let's say you wish to call the Visible Monitor from BASIC. The Visible Monitor's entry point is at $1207, so we must make $000B,000C point to $1207. This means storing 07 in $000B, and storing $12 (decimal 18) in $000C. The following line will do that for us:

```
POKE 11,7:POKE 12,18
```

Now we may invoke the Visible Monitor with the line:

```
X = USR(X)
```

or with any other line that uses the USR function.

Note that the USR function does not set a BASIC variable equal to the contents of some register in the 6502; in fact, the line X = USR(X) will not change the value of the BASIC variable X at all. Thus, the USR function lets you activate any desired machine-language subroutine, but it doesn't let you capture a value returned by such a subroutine. If you want a machine-language subroutine to return some value which you can then use in a BASIC program, you'll have to make the machine-language subroutine store its value or values somewhere in memory, and then have the BASIC program PEEK that memory location after it has called the machine-language subroutine via the USR function.

Appendix B2:

The PET 2001

Display Memory

The PET screen is mapped conventionally, with the HOME address at $8000 (32,768 decimal). It has 25 rows, each consisting of 40 characters. The address of each screen location is 40 ($28) greater than the address of the screen location directly above it. Thus, the screen parameters for the PET 2001 are:

```
HOME      .WORD $8000,
ROWINC    .BYTE $28
TVCOLS    .BYTE 39          (We count columns from zero.)
TVROWS    .BYTE 24          (We count rows from zero.)
```

PET Character Set

However, although the PET screen buffer is mapped conventionally, you cannot simply store an ASCII character in screen memory if you wish to see that ASCII character on the screen. The PET character generator introduces a few wrinkles and you must compensate carefully if you are to display ASCII characters properly on the screen.

For example, if you store $31 (the code for an ASCII "1") in the PET's display memory, then you will see a "1" displayed on the screen. So far, so good. The same is true for all ASCII digits and for some ASCII punctuation marks. But if you store $45 (ASCII code for an upper case "E") in screen memory, then you won't see an "E" on the screen: you'll see either a lowercase "e" or else a horizontal line segment much longer than a hyphen. What's happening?

The PET 2001 features a memory location, $E84C (59468) which has a special effect on the video-display circuitry. The value stored in that address selects for the video display one character set or another.

To see how the choice of character set affects the display, enter the following BASIC program into your PET:

```
100  REM       DISPLAY PET CHARACTER SET
110  REM       IN 16 BY 16 MATRIX
120  REM
130  HOME=32768
140  CHAR=0
150  FOR ROW=0 TO 15
160  FOR COL=0 TO 15
170  POKE (HOME+COL)+(40*ROW),CHAR
180  CHAR=CHAR+1
190  NEXT COL
200  NEXT ROW
210  END
```

Before running this program, clear the screen by holding down the PET's SHIFT key at the same time that you depress the CLR/HOME key. When the screen is clear, use the CRSR SOUTH key to move the cursor down seventeen rows. Then type RUN and press RETURN. You'll see one PET character set appear in a 16 by 16 matrix in the upper left portion of your PET's screen.

What you'll see on your screen will look like table B2.1 (without the labeled axes).

Table B2.1: *The PET character set.*

LEFT NYBBLE OF CHARACTER	RIGHT NYBBLE OF CHARACTER −0	−1	−2	−3	−4	−5	−6	−7	−8	−9	−A	−B	−C	−D	−E	−F
0−	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1−	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
2−		!	"	#	$	%	&	'	(	)	*	+	'	—	.	/
3−	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4−	—	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
5−	p	q	r	s	t	u	v	w	x	y	z	—	—	—	—	—
6−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
7−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
8−	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9−	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	←
A−		!	"	#	$	%	&	'	(	)	*	+	'	—	.	/
B−	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
C−	—	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
D−	p	q	r	s	t	u	v	w	x	y	z	—	—	—	—	—
E−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
F−	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

In this chart, special graphic characters are indicated by an underline. Look at your PET screen to see those special graphics in all their glorious detail.

Note that the characters for $80 thru $FF are the same as for $00 thru $7F, but in reverse intensity. The low 128 characters ($00 thru $7F) are "normal" — that is, white characters on a dark background; whereas the high 128 characters ($80 thru $FF) are in reverse video — dark characters in a white background. An "A" in normal intensity may be displayed by storing an $01 somewhere in the screen memory; a reverse intensity "A" may be displayed by storing an $81 somewhere in screen memory. From this pattern we can derive a handy corollary: to reverse the intensity of any character on the screen, simply reverse its bit 7. You don't even have to know what the character represents; just toggle bit 7 and you change its intensity.

The chart in figure B2.1 (and on your PET screen) shows one complete character set because the BASIC program stores every 8-bit value, from $00 thru $FF, into the screen buffer. But I mentioned two character sets. What must you do to see the second character set?

If the cursor is within three rows of the bottom of the screen, move it up so that it is at least three rows above the bottom of the screen. This will insure that you don't scroll part of the character set up off the screen when you execute the following BASIC command in the immediate mode:

```
POKE 59468,12
```

Did that change the display? If not, then execute the following BASIC command in the immediate mode (again being sure that the cursor is at least three rows from the bottom of the screen):

```
POKE 59468,14
```

Depending on the value stored in 59468 ($E84C), one or another character set will be displayed. The values of the bytes stored in screen memory will not change when you change the contents of $E84C, but in some cases the displayed characters will change. In the ranges 00 thru $3F and $80 thru $BF, the two character sets are identical. But in the ranges $40 thru $7F and $C0 thru $FF, they differ.

Both character sets include numbers, uppercase letters, and certain punctuation marks; but only one character set includes lowercase letters and the remaining punctuation marks. The second character set lacks lowercase letters and these punctuation marks, offering instead a set of special graphics characters, including playing-card suits. POKE 59468,14 to select the former character set (thereby making possible the display of all printable ASCII characters); POKE 59468,12 to select the latter character set (thereby making possible the display of the gaming graphics).

FIXCHR

Note that neither character set corresponds directly to ASCII. If you have an ASCII character in the accumulator and you want to display the appropriate graphic character on the screen, you must first call FIXCHR (as TV.PUT does, in Chapter 5). When an ASCII character is passed in the accumulator, FIXCHR must return in the accumulator the proper PET display code for that character. FIXCHR's caller may then store this display code in memory, thereby placing on the screen an appropriate image of the original ASCII character.

How will FIXCHR work? By examining the PET character set and comparing it to Appendix A2, ASCII codes, we can see a solution in the form of the following algorithm:

- If a character is in the range $40 thru $5F, subtract $40 and return.
- If a character is in the range $20 thru $3F, return.
- If a character is in the range $60 thru $7A, store a decimal 14 in 59468 to select the character set that has lower case letters; and return.
- All other input characters are either ASCII control codes, for which there are no agreed-upon graphics, or else PET special graphics characters, so just return.

Examine the tables yourself to see if this algorithm will work.

FIXCHR

```
FIXCHR   AND #$7F        Clear bit 7, so the character will be in
                         the legal ASCII range.
         SEC             Prepare to compare.
         CMP #$40        If it's less than $40, return.
         BCC FIXEND
                         Okay. The character is greater than $40.
         CMP #$60        Is it greater than $5F?
         BCS LOWERC      If so, handle it as lowercase.
                         Okay. The character is in the range
                         $40-$5F.
         SBC #$40        Subtract $40 for proper display code.
         RTS
LOWERC   LDX #14         Since we have a lowercase letter, let's
                         select the character set that
         STX 59468       has lowercase letters.
FIXEND   RTS             Return, bearing PET display code for
                         character originally in accumulator.
```

Call FIXCHR with an ASCII character in the accumulator. FIXCHR will return with the corresponding PET display code in the accumulator. When it returns, its caller may store the accumulator anywhere in screen memory, thus displaying an image of the original ASCII character.

PET Keyboard Input Routine

To get an ASCII character from the PET keyboard, call the following subroutine:

```
PETKEY   JSR $FFE4      Call PET ROM key scan routine.
         CMP #0         Zero means no key.
         BEQ PETKEY     If no key, scan again.
                        A new key is in the accumulator. If the
                        shift key was down, bit 7 is set.
         AND #$7F       So clear bit 7, just to be sure we've got
                        a legal ASCII character.
         RTS            Return with ASCII character in the ac-
                        cumulator.
```

This subroutine yields the uppercase ASCII code for any letter key that you depress, and the proper ASCII code for any digit key or punctuation key.

PET TVT Routine

To print an ASCII character to the screen, call $FFD2, a PET ROM routine I will refer to as PETTVT.

Any printable ASCII character passed to $FFD2 (or, apparently, to $E3EA or $F230) will be printed properly to the screen at the PET's current TVT screen location. You may change the PET's current TVT screen location (which is *not* the same as the current location used by the screen utilities in Chapter 5) by calling PETTVT with the accumulator holding any of the control codes from Table B2.1.

Table B2.1: *Control codes that affect the next character to be printed by PETTVT.*

Character Name	Code	Function
CURSOR NORTH	$91	Move current location up by one row.
CURSOR EAST	$1D	Move current location one column to the right.
CURSOR SOUTH	$11	Move current location down by one row.
CURSOR WEST	$9D	Move current location left by one column.
INSERT	$94	Move current character, and all characters to its right, one column to the right.
DELETE	$14	Move current character, and all characters to its right, one column to the left.
HOME	$13	Set current location to upper left of screen.
CLEAR	$93	Set current location to the upper left corner and clear the screen.
REVERSE	$12	Select reverse video for following characters.
REVERSE-OFF	$92	Select normal video mode for following characters.

These control codes may be passed directly to PETTVT, or they may be included within a string of characters to be printed by "PRINT:" or "PR.MSG." For example, if you wish to clear the screen before printing a message, just put the CLEAR character ($93) at the beginning of your message string, immediately following the STX. The message-printing subroutine will get the CLEAR character and pass it to PR.CHR, which, in turn, will pass it through the ROMTVT vector on to the PETTVT routine. The PETTVT routine will then clear the screen and set the current location to the upper left corner of the screen.

The next character in the string will then be printed in the upper left corner of a clear screen. If, instead of printing your message at the top row of a clear screen, you'd prefer to print it in the fifth row of a clear screen, just follow the CLEAR character with four CURSOR-SOUTH characters ($11, $11, $11, $11), and follow the four cursor-south characters with the text of your message. Following the text of your message, of course, you must include an ETX ($FF).

You might never use the PETTVT control codes, but it's good to know they're available, should you ever want your PET's display screen to perform as something more than a glass teletype.

System Data Block

To run on a PET 2001, the software in this book requires the system data block shown in Appendices C14 and E13.

Setting the Top of Memory

Before you can use the BASIC OBJECT CODE LOADER (presented in Chapter 12) to load object code into your PET's memory, you must insure that your PET's BASIC interpreter leaves undisturbed all memory above $0FFF (4095 decimal). The PET BASIC interpreter will do as we wish if we set its top-of-memory pointer appropriately. The top-of-memory pointer specifies the highest address that may be used for the storage of BASIC program lines, variables, and strings. Memory above that address is off-limits to BASIC.

As you may know, there is more than one version of the PET 2001 by Commodore. Some PET's have software in "old" ROMS (REV 2 ROMS), and others have software in "new" ROMS (REV 3 ROMS). As far as the software in this book is concerned, old ROM PETS and new ROM PETS are the same, since the ROM routines we care about are accessible from the same addresses in both old and new ROM PETS. Therefore, until now I haven't even mentioned that the PET 2001 comes in two flavors. But now you must discover whether you have an old ROM or a new ROM PET, because otherwise you won't be able to set the top of memory.

Old ROM and new ROM PETS each contain a machine-language subroutine to clear the screen, but in new ROM PETS that subroutine is at $E229 (57897 decimal), and in old ROM PETS that subroutine is as $E236 (57910 decimal). To see what ROMS are in your PET, use the PET's screen editor to place some characters on the screen, and then type:

```
SYS (57897)
```

and press (RETURN). Does the screen clear? If so, you've got a new ROM PET. If not, turn off your PET, turn it on, place some characters on the screen, and then type:

```
SYS (57910)
```

and press (RETURN). Does the screen clear? If so, you've got an old ROM PET. If not, then your PET contains neither Rev 2 ROMS nor Rev 3 ROMS, and you'll have to consult your system's documentation carefully to discover the address of the top-of-memory pointer.

On old ROM PETS, the top-of-memory pointer is at 134 and 135 ($86,87). On new ROM PETS, the top-of-memory pointer is at 52 and 53 ($34,35). Regardless of the location of the top-of-memory pointer, we want to set the low byte of that pointer equal to $FF (255 decimal), and the high byte of that pointer equal to $0F (15 decimal), so that the pointer itself points to $0FFF. That will leave memory from

$1000 and up available to machine-language programs.

Thus, we set the top of memory on an old ROM PET with:

POKE 134,255:POKE 135,15

Similarly, we set the top of memory on a new ROM PET with:

POKE 34,255:POKE 35,15

Once you have set the top of memory available to your PET's BASIC interpreter, you may enter the BASIC OBJECT CODE LOADER and the DATA statements from Appendices E1 thru E11, and from Appendix E13. Remember to set the top of memory not only when typing in these DATA statements, but when RUNning the OBJECT CODE LOADER, as well.

Appendix B3:

The Apple II

Apple Display

The display memory of the Apple II is mapped in a manner that is much more complex than the Ohio Scientific or PET computers. On each of these other systems, only one portion of memory is mapped to the screen. The screen cannot display the contents of any other bank of memory (unless, of course, you copy the contents of another bank of memory into the display memory). But the Apple II may display the contents of any of four banks of memory: Low-Resolution Graphics and Text Page 1, Low-Resolution Graphics and Text Page 2, High-Resolution Graphics Page 1, and High-Resolution Graphics Page 2. Table B3.1 summarizes the locations of these pages in memory.

Table B3.1: *Banks of display memory in the Apple II.*

	Hexadecimal	**Decimal**
Low-Resolution Graphics and Text Page 1:	\$0400-\$07FF	1024-2043
Low-Resolution Graphics and Text Page 2:	\$0800-\$0BFF	2048-3071
Hi-Resolution Graphics Page 1:	\$2000-\$3FFF	8192-16383
Hi-Resolution Graphics Page 2:	\$4000-\$5FFF	16384-24575

Note that each of these display pages takes up much more than one hexadecimal page (256 bytes). A display page is simply an area of any size memory, whose contents may be displayed on the screen. Each low-res display page occupies four hexadecimal pages, and each hi-res display page occupies 32 hexadecimal pages. Why are the hi-res display pages bigger than the low-res display pages? Hi-res means high-resolution, and higher resolution requires more information.

How do you make the video screen show the contents of a given display page? You need only store a zero in a particular address. Certain addresses in the Apple II signal the video-display circuitry whenever data are written to them. The video-display circuitry responds to these signals by displaying the contents of a given bank of memory. These special addresses, or *display selectors*, are given in Table B3.2.

Table B3.2: *Addresses that affect the APPLE II Display.*

Hexadecimal	Decimal	Label	Purpose of Address
$C050	−16304	TXTCLR	Store a 0 here to set graphics mode.
$C051	−16303	TXTSET	Store a 0 here to set text mode.
$C052	−16302	MIXCLR	Store a 0 here to set bottom four lines to graphics.
$C053	−16301	MIXSET	Store a 0 here to select text/graphics mix (bottom four lines text).
$C055	−16299	HISCR	Store a 0 here to select Page 2.
$C056	−16298	LORES	Store a 0 here to select low-resolution graphics and text page.
$C057	−16297	HIRES	Store a 0 here to select high-resolution graphics.

Space limitations prohibit a discussion in this book of the power of high-resolution graphics. The Apple II documentation, however, provides an excellent step-by-step guide to the design, display, saving, and loading of high-resolution images. I must stress, however, that the software in this book expects the host system to have low-resolution graphics, so you'd better tell your Apple II to have low-resolution graphics. The software in this book uses the Apple's low-resolution graphics with text page 1 as the screen memory. To select this display page, simply press the RESET button on your Apple. If, on the other hand, you wish to select this display page under software control, you can do it by calling the subroutine LORES1:

```
LORES1   PHP               Save processor flags.
         PHA               Save accumulator.
         LDA # 0           Store a 0 in
         STA LOWSCR        LOWSCR to select Page 1,
         STA LORES         and in LORES to select low-resolution
                           graphics.
         PLA               Restore accumulator.
         PLP               Restore processor flags.
         RTS               Return to caller.
```

This subroutine will select low-resolution graphics and text page 1. It preserves all flags and registers, and is completely relocatable.

Even when you've configured your Apple II to low-resolution graphics, your job isn't done. The low-res display of the Apple II is mapped in an unusual manner. For any other system you can assume that the address of a given location on the screen is simply the address of the location above it, plus some row increment. On the Apple II this is not always true. See Table B3.3, Apple II low-res display memory map.

Table B3.3: *Apple II low-resolution display.*

Page 1

Row Number	Address of Leftmost Column	Address of Rightmost Column
\$00	\$400	\$427
\$01	\$480	\$4A7
\$02	\$500	\$527
\$03	\$580	\$5A7
\$04	\$600	\$627
\$05	\$680	\$6A7
\$06	\$700	\$727
\$07	\$780	\$7A7
\$08	\$428	\$44F
\$09	\$4A8	\$4CF
\$0A	\$528	\$54F
\$0B	\$5A8	\$5CF
\$0C	\$628	\$64F
\$0D	\$6A8	\$6CF
\$0E	\$728	\$74F
\$0F	\$7A8	\$7CF
\$10	\$450	\$477
\$11	\$4D0	\$4F7
\$12	\$550	\$577
\$13	\$5D0	\$5F7
\$14	\$650	\$677
\$15	\$6D0	\$6F7
\$16	\$750	\$777
\$17	\$7D0	\$7F7

Page 2

Row Number	Address of Leftmost Column	Address of Rightmost Column
$00	$800	$827
$01	$880	$8A7
$02	$900	$927
$03	$980	$9A7
$04	$A00	$A27
$05	$A80	$AA7
$06	$B00	$B27
$07	$B80	$BA7
$08	$828	$84F
$09	$8A8	$8CF
$0A	$928	$94F
$0B	$9A8	$9CF
$0C	$A28	$A4F
$0D	$AA8	$ACF
$0E	$B28	$B4F
$0F	$BA8	$BCF
$10	$850	$877
$11	$8D0	$8F7
$12	$950	$977
$13	$9D0	$9F7
$14	$A50	$A77
$15	$AD0	$AF7
$16	$B50	$B77
$17	$BD0	$BF7

Note that the display addresses do not increase uniformly as we move down, row-by-row, through low-res display page 1 or 2. The addresses increase uniformly from row 0 thru row 7, but from row 7 to row 8 the display addresses do not increase; they decrease! Then they increase uniformly through line $0F (15 decimal), but from line $0F to line $10 (15 to 16 decimal), the display address plummets again. Then from row $10 to row $17 (16 thru 23) the display addresses again increase uniformly.

If you'd like to take a visual tour of the Apple II's low-res display memory, run the BASIC program in listing B3.1. This program will simply poke a blank into each address in low-res display page 1, starting at the lowest address and moving to the highest address. You'll see that the screen does not fill with blanks in a contiguous manner, but follows a pattern of three interleaved parts.

Listing B3.1: *APPLE II low-resolution display, memory-mapper program.*

```
100   REM APPLE II LOW-RESOLUTION DISPLAY, MEMORY-MAPPER
105   REM
108   REM BY KEN SKIER
110   REM
120   FIRST=1024: REM START OF LOW-RESOLUTION PAGE 1.
130   LAST=2043: REM  END OF LOW-RESOLUTION PAGE 1.
140   CHAR=32: REM    CHARACTER TO BE POKED INTO SCREEN
150   REM             WILL BE A WHITE BLANK.
160   REM
170   FOR X=FIRST TO LAST
175   REM   FOR EACH ADDRESS IN LOW-RESOLUTION PAGE 1.
180   POKE X,CHAR
185   REM             POKE A WHITE BLANK. THEN,
190   GOSUB 1000: REM WAIT A MOMENT...
200   NEXT X: REM     BEFORE POKING NEXT ADDRESS.
210   END
220   REM
230   REM
1000  FOR WAIT=0 TO 100
1005  REM             THIS IS A WAIT SUBROUTINE.
1010  NEXT WAIT: REM IT SLOWS DOWN PROGRAM SO YOU
1020  RETURN: REM     CAN FOLLOW THE ACTION.
```

Must we now write a whole new set of display procedures to accommodate the unusual mapping of the Apple II low-res display pages? We could. But the screen utilities presented in Chapter 5 will work for the Apple II if we think of the Apple low-res screen as three separate screens: the top eight rows are one screen, the middle eight rows are another screen, and the bottom eight rows are a third screen. Each of these "screens" has a set of screen parameters.

The screen utilities in this book will work fine if you limit their scope to a given third of the screen. Use TVTOXY only to set a relative screen position within the third of the screen that you have selected. Use the screen utilities only for the top third of the screen. The middle and bottom thirds of the screen may still be used by the PRINT utilities.

To limit the screen utilities to the top third of low-res display page 1, initialize the screen parameters as follows:

```
SCREEN    .WORD $0400
TVCOLS    .BYTE $27
TVROWS    .BYTE $07
ROWINC    .BYTE $80
```

If you want to keep text from scrolling into the upper third of the screen, store $08 in address $0022. (In BASIC you may do this with the command POKE 34,8.)

There's one more quirk to the Apple display. If you store an ASCII character in display memory, then you will display a blinking or inverse version of the character. Setting bit 7 in an ASCII character code will cause that character to be displayed in normal mode (a white character on a black background), rather than as a black character on a white background or as a blinking character.

You may experiment with this feature of the Apple II by using the Apple II monitor to store $41 (an ASCII "A") in a location in low-res display page 1. You'll see a blinking "A." Now store $C1 in a location in low-res display page 1. You'll see a normal "A." Why? Because $C1 is $41 with bit 7 set. To understand what's happening here, look at the Apple II's character set given in Table B3.4.

Table B3.4: *The Apple II character set.*

LEFT NYBBLE OF CHARACTER	RIGHT NYBBLE OF CHARACTER —0	—1	—2	—3	—4	—5	—6	—7	—8	—9	—A	—B	—C	—D	—E	—F
0—	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1—	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]		--
2—		!	"	#	$	%	'	(	)	*	+	,	—	.	/	
3—	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4—	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5—	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]		--
6—		!	"	#	$	%	'	(	)	*	+	,	—	.	/	
7—	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
8—	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
9—	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]		--
A—		!	"	#	$	%	'	(	)	*	+	,	—	.	/	
B—	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
C—	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
D—	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]		--
E—		!	"	#	$	%	'	(	)	*	+	,	—	.	/	
F—	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

The Apple II really has only 64 characters in its character set, but it has four ways of displaying each character. Thus, the table shows a set of characters at $00 thru $3F; the same characters, in the same sequence, appear again at $40 thru $7F, at $80 thru $BF, and at $C0 thru $FF. These represent what I call the first, the second, the third, and the fourth quadrants of the character set.

Character codes in this first quadrant ($00 thru $3F) will be displayed in reverse video: as black characters on a white background. Character codes in the second quadrant ($40 thru $7F) will be displayed in a blinking mode. Character codes in the third and fourth quadrants ($80 thru $BF and $C0 thru $FF) will be displayed in normal mode: as white characters on black background.

Before we store any ASCII character in screen memory, we must first call FIXCHR, to convert, if necessary, the ASCII character to the host system's corresponding display code. In the Apple II, FIXCHR is very simple:

FIXCHR	ORA #$80	Set bit 7, so character will be displayed in normal mode.
RTS		Return appropriate display code to caller.

I/O Vectors

The Apple II has a subroutine in read-only memory to get a character from the keyboard, and another subroutine to print a character on the screen. However, the key-in routine at $FD35 does not return an ASCII code when you press the key for an ASCII character; instead, it returns the appropriate ASCII code with bit 7 set. Similarly, the screen-printing routine at $FBFD will print an ASCII character to the screen, but the character will be in reverse video or blinking. In order to print an ASCII character to the screen, you must first set bit 7 and then call $FBFD. Conversely, to get an ASCII character from the keyboard, you must first call $FD35 and then clear bit 7. Therefore, the following patches are offered:

Subroutine to Print an ASCII Character to Apple II Screen

APLTVT	ORA #$80	Set bit 7 in the ASCII code.
	JSR $FBFD	Call the ROM screen printer.
	RTS	Return to caller, now that ASCII character originally in accumulator has been printed to screen in normal mode.

Subroutine to Get an ASCII Character from Apple II Keyboard

APLKEY	JSR $FD0C	Get ASCII character from keyboard with bit 7 set. (Note: you may call $FD35 instead of calling $FD0C.)

ORA #$80	Clear bit 7, leaving the accumulator holding a conventional ASCII code.
RTS	Return to caller, bearing ASCII character code for depressed key.

Apple II System Data Block

The I/O vectors ROMTVT and ROMKEY should be initialized to point to APLTVT and APLKEY, respectively. This has been done in the Apple II system data block. You *must enter* the Apple II system data block into your system's memory if any of the software in this book is to run on your Apple II. See Appendices C15 and E14.

Appendix B4:

The Atari 800

Screen

The Atari 800 microcomputer has the most flexible — and, perhaps the most confusing — video-display hardware of any system discussed in this book. Unlike the other systems, almost any portion of the Atari computer's memory may be mapped to the screen. Furthermore, there are many different screen-display modes. When the Atari computer is powered-up, the screen is in text mode zero. That's comparable to the Apple II's low-resolution graphics and text display, which is comparable to the only video-display mode available on the Ohio Scientific or PET computers.

The Atari computer makes other screen modes available to the programmer, but the software in this book assumes a low-resolution text display, so you'd better leave your Atari in screen mode zero if you expect to see any of the displays driven by the software in this book. In other words, if you change the screen mode, the Visible Monitor may well become invisible.

I mentioned that the screen buffer may be almost anywhere in memory. If that's true (and it is), how can you determine the HOME address upon which all the displays in this book are based? It's easy. A pointer at $58,$59 (88,89 decimal) points to the lowest address in screen memory: the address we refer to as HOME. Before running any of the software in this book, you must set HOME properly for your system. Simply set HOME equal to the value of that pointer. HIPAGE, the value of the highest page in screen memory, is equal to (the high byte of HOME) plus three.

Once we've set HOME and HIPAGE properly, we're home free. The other screen parameters are fixed:

```
ROWINC   .BYTE 40
TVCOLS   .BYTE 39
TVROWS   .BYTE 23
SPACE    .BYTE $20
ARROW    .BYTE $7B
```

Note that the top of screen memory is always at the top of programmable memory, so if you add more programmable memory to your Atari 800, you'll move the screen memory up higher in the address space.

Proper Display of ASCII Characters

Like the PET, and to a lesser extent the APPLE II, the Atari screen requires that we perform a conversion before we can properly display an ASCII character on the screen. To determine the nature of this conversion, let us first look at the ATARI character set in Table B4.1.

Table B4.1: *The Atari character set ATASCI.*

	−0	−1	−2	−3	−4	−5	−6	−7	−8	−9	−A	−B	−C	−D	−E	−F
0−	*space*	!	"	#	$	%	&	'	(	)	*	+	,	−	.	/
1−	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
2−	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3−	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]		←
4−	*special graphics characters*															
5−	*special graphics characters*															
6−		a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7−	p	q	r	s	t	u	v	w	x	y	z	*graphics*				

A quick examination shows that ASCII characters $20 thru $5F are ATASCI (Atari's character set) characters $00 thru $3F. Thus, if an ASCII character is in the range of $20 thru $5F, we can convert it to the appropriate ATASCI character simply by subtracting $20.

Further inspection reveals that ASCII characters $61 thru $7A correspond to ATASCI characters $61 through $7A. Thus, if an ASCII character is in the range of $61 thru $7A, it needs no conversion to ATASCI; it already *is* the corresponding ATASCI character.

Finally, if an ASCII character is not in the range $20 thru $5F or $61 thru $7A, it's not a printable character and has no agreed-upon graphic representation. For those cases we'll just leave them alone.

Figure B4.1 flow-charts this algorithm.

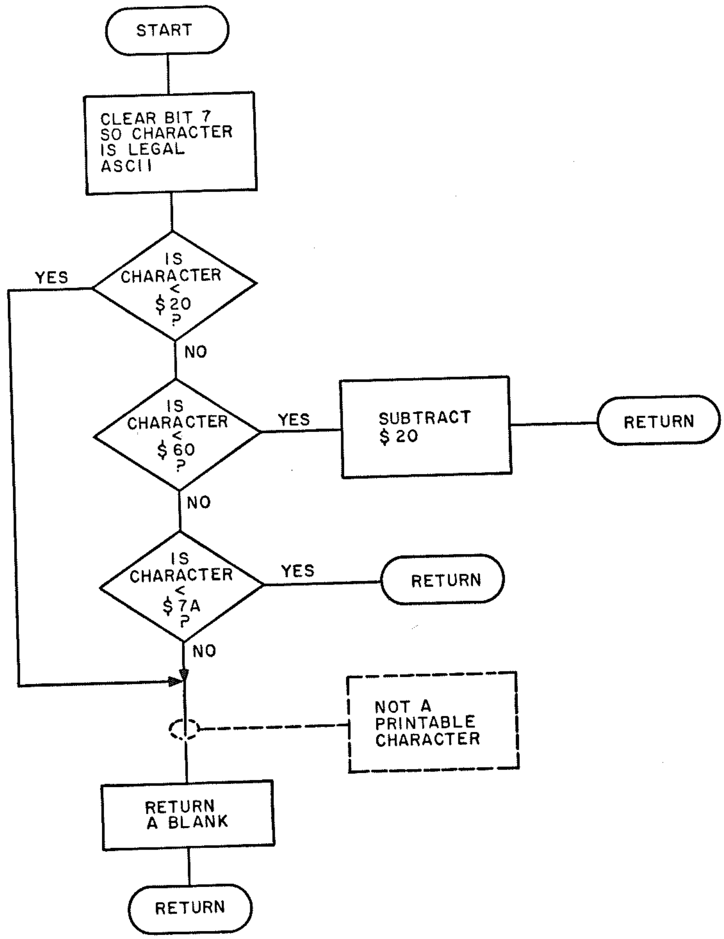

Figure B4.1: *Flowchart of routine to convert an ASCII character for display on Atari screen.*

Using the flowchart in figure B4.1 as a guide, we can write source code for FIXCHR, which takes an ASCII character as input and returns an Atari display code so that the character may be properly displayed on the video screen.

FIXCHR

```
FIXCHR    AND #$7F      Clear bit 7 so character is a legitimate
                        ASCII character.
          SEC           Prepare to compare.
          CMP #$20      Character less than $20?
```

```
        BCC BADCHR        If so, it's not a printable ASCII
                          character, so return a blank.
        CMP #$60          Character less than $60?
        BCC SUB$20        If so, subtract $20 and return.
        CMP #$7B          Character less than $7B?
        BCC EXIT          If so, return with the character.
                          If not less than $7B,
BADCHR  LDA BLANK         the character is not a printable ASCII
                          character, so return a blank.
EXIT    RTS
SUB$20  SBC #$20          Subtract $20 and
        RTS               return.
```

Keyboard Input

If no key has been pressed, then address $02FC (764 decimal) contains $FF. But whenever you depress a key on the Atari keyboard — even if a program is not scanning the keys — an electronic circuit will sense that a key has closed and will store the hardware code for that key in address $02FC. However, the code in $02FC will be a hardware code, not obviously related to ASCII or ATASCI.

Table B4.2: *Atari Hardware Key-Codes.*

Hex	Decimal	Key	Hex	Decimal	Key
$00	0	L	$20	32	,
1	1	J	1	33	SPACE
2	2	;	2	34	.
3	3		3	35	N
4	4		4	36	
5	5	K	5	37	M
6	6	+	6	38	/
7	7	*	7	39	ATARI
8	8	0	8	40	R
9	9		9	41	
A	10	P	A	42	E
B	11	U	B	43	Y
C	12	RETURN	C	44	TAB
D	13	I	D	45	T
E	14	−	E	46	W
F	15	=	F	47	Q

$10	16	V	$30	48	9
1	17		1	49	
2	18	C	2	50	Ø
3	19		3	51	7
4	20		4	52	BACK S
5	21	B	5	53	8
6	22	X	6	54	<
7	23	Z	7	55	>
8	24	4	8	56	F
9	25		9	57	H
A	26	3	A	58	D
B	27	6	B	59	
C	28	ESC	C	60	LOWR
D	29	5	D	61	G
E	30	2	E	62	S
F	31	1	F	63	A

The Hex and Decimal Columns give the low 6 bits of the hardware key-code stored in address $02FC (764 decimal) when the given keys are pressed. Either SHIFT key sets bit 6. CTRL key sets bit 7.

In order to convert that hardware code to ASCII, we need to understand its nature. The six low-order bits of the hardware key-code uniquely identify the key. (See Table B4.2.) Bits 6 and 7 identify its shift state. Bit 6 is set if the key is typewriter-shifted; bit 7 is set if the key is control-shifted. The key is typewriter-shifted if either SHIFT key is down; the CAPS/LOWR key has no effect on the typewriter-shift state as reflected in the hardware key-code. The keyboard is control-shifted if the CTRL key is down.

If you don't care about the keyboard's shift state, but merely want to determine which physical key has been pressed, then you can clear the two high-order bits in the hardware key-code and you'll be left with a number from 0 to 63 decimal (00 to $3F) uniquely identifying the key most recently depressed. If you care about the keyboard's typewriter-shift state but are indifferent to its control-shift state, then you can clear bit 7 in the hardware key-code and you'll be left with a number from 0 to 127 decimal (00 to $7F), which means the keyboard can generate twice as many characters as it has physical keys. To enable control-shifting, simply preserve the hardware key-code, and you double once again the number of characters that the keyboard (and hence the user) may generate.

Since the simple text editor presented in Chapter 11 assigns certain functions to control-shifted keys, and since you never know when you might need some additional character codes from your keyboard, Appendix C16 presents a key-handling subroutine for the Atari. This subroutine is capable of generating different characters in each of the four different shift-states (unshifted, typewriter-shifted, control-shifted, typewriter- and control-shifted).

It's a simple matter to use the eight-bit hardware keycode as an index into a keyboard definition table. For any given hardware key-code, we may assign any character we like. The keyboard definition table presented in Appendix C16 assigns standard ASCII characters to all letter, number, and punctuation keys, in both the unshifted and typewriter-shifted states. Other keys are assigned values consistent with their expected use by the software in this book (eg: Control-P generates a $10, thus making it a PRINT key in the eyes of the simple text editor). All keys and shift states that have no special meaning to this software have been assigned character codes of zero; feel free to change these character codes to any values you desire.

Assuming that we have in memory a keyboard definition table called ATRKYS, we can get an ASCII character from the Atari keyboard with the following subroutine, ATRKEY:

```
ATRKEY   LDA $02FC        Has a key been depressed?
         CMP #$FF         $FF means no key.
         BEQ ATRKEY       If not, look again. A key has gone down
                          and the accumulator holds its hardware
                          key-code.
         TAY              Prepare to use that code as an index.
         LDA ATRKYS,Y     Look up character for that key and shift
                          state.
         RTS              Return with ASCII character
                          corresponding to that key and shift
                          state.
```

Print a Character to the Screen

The Atari 400 and 800 computers each provide a powerful I/O (input/output) routine which allows the programmer to get characters from virtually any source, and to send characters to virtually any device — the screen, the printer, the cassette recorder, and the disk. But, as in the case of Atari's varied screen modes, power breeds complexity. I have found it easier to substitute my own simple routine to print a character on the TV screen, bypassing the Atari I/O routines entirely.

Incidentally, this routine will work with any 6502-based computer that has a low-resolution memory-mapped display. If you need a simple TVT simulator for your home-brew 6502-based system with a video display, TVTSIM might meet your needs. In any event, it prints characters to the screen, and avoids the necessity of plumbing the depths of the many modes and data structures associated with Atari's central I/O routine.

With your system data block initialized as shown in Appendices C16 and E15 (which includes the TVT simulator as the subroutine to print characters to the screen), you are almost ready to run the software in this book on your own system.

Setting the Top Of Memory

Address $2E6 (742 decimal) holds the number of pages of RAM available to the BASIC interpreter. Store a $0D (13 decimal) in that location and BASIC will use memory up to $0DFF, but will not use $0E00 and up.

NOTE: On the Atari, the software in this book uses memory from $0E80 to $1FFF, which is the address space required by the ATARI DOS (Disk Operating System) and the ATARI RS-232 serial interface, so you may *not* use DOS or RS-232 if you expect to use the software in this book. However, there should be no conflict between software in this book and the cassette-based Atari 800.

Thus, we may set the top of memory with the following BASIC command:

```
POKE 742,13
```

When you have used the OBJECT CODE LOADER to READ and POKE object code from all the appropriate E appendices into your Atari computer, run the following BASIC program. It will initialize screen parameters and the top of memory, and then pass control to the Visible Monitor.

```
100 REM               Visible Monitor Start-Up Program for the Atari.
110 REM
120 REM               First, set the screen parameters.
130 REM
140 REM               A pointer at 88,89 points to lowest screen address.
150 LO=PEEK(88):      REM Set LO to the low byte of HOME.
160 HI=PEEK(89):      REM Set HI to the high byte of HOME.
165 IF HI < 32 THEN PRINT "ON AN 8 K ATARI YOU MAY NOT USE EDITOR
    OR DISASSEMBLER"
170 POKE 4096,LO:     REM Set Low byte of HOME.
180 POKE 4097, HI:    REM Set High byte of HOME.
190 POKE 4101,HI+3:   REM Set HIPAGE = Highest page in screen memory.
200 REM
210 REM               Now set the top of memory available to BASIC.
220 POKE 742,13:      Tell BASIC to use only memory up to $0DFF.
230 REM
240 REM               Now call the Visible Monitor.
250 X=USR(4615):      REM Call the Visible Monitor as a subroutine.
260 END
```

Appendix C1:

Screen Utilities

```
                  ;     APPENDIX C1: ASSEMBLER LISTING OF
                  ;                  SCREEN UTILITIES
                  ;
                  ;
                  ;
                  ;       SEE CHAPTER 5 OF BEYOND GAMES: SYSTEMS
                  ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                  ;
                  ;
                  ;                     BY KEN SKIER
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; *******************************************
                  ;
                  ;         ZERO PAGE BYTES
                  ;
                  ; *******************************************
                  ;
                  ;
                  ;
                  ;
                  ;
    0000=               TV.PTR=0       THIS POINTER HOLDS THE
                  ;                    ADDRESS OF THE CURRENT
                  ;                    SCREEN LOCATION.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; *******************************************
                  ;
                  ;         SCREEN PARAMETERS
                  ;
                  ; *******************************************
                  ;
                  ;
                  ;
    1000=               PARAMS=$1000   THE FOLLOWING ADDRESSES
                  ;                    MUST BE INITIALIZED TO HOLD
                  ;                    DATA DESCRIBING THE SCREEN
                  ;                    ON YOUR SYSTEM.
                  ;
                  ;
                  ;
                  ;
    1000=               HOME=PARAMS    HOME IS A POINTER TO CHARACTER
```

```
                  ;                      POSITION IN UPPER LEFT CORNER.
                  ;
     1002=             ROWINC=PARAMS+2
                  ;                      ROWINC IS A BYTE GIVING
                  ;                      ADDRESS DIFFERENCE FROM ONE
                  ;                      ROW TO THE NEXT.
                  ;
     1003=             TVCOLS=PARAMS+3
                  ;                      TVCOLS IS A BYTE GIVING
                  ;                      NUMBER OF COLUMNS ON SCREEN.
                  ;                      (COUNTING FROM ZERO.)
                  ;
     1004=             TVROWS=PARAMS+4
                  ;                      TVROWS IS A BYTE GIVING
                  ;                      NUMBER OF ROWS ON SCREEN,
                  ;                      (COUNTING FROM ZERO.)
                  ;
     1005=             HIPAGE=PARAMS+5
                  ;                      HIPAGE IS THE HIGH BYTE OF
                  ;                      THE HIGHEST ADDRESS ON SCREEN.
                  ;
                  ;
     1006=             BLANK=PARAMS+6 YOUR SYSTEM'S CHARACTER
                  ;                      CODE FOR A BLANK.
                  ;
     1007=             ARROW=PARAMS+7 YOUR SYSTEM'S CHARACTER
                  ;                      FOR AN UP-ARROW.
                  ;
     1011=             FIXCHR=PARAMS+$11
                  ;                      FIXCHR IS A SUBROUTINE THAT
                  ;                      RETURNS YOUR SYSTEM'S
                  ;                      DISPLAY CODE FOR ASCII.
                  ;                      CODE.
                  ;
                  ;
                  ;
                  ;
                  ;
     1100              *=$1100
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;          CLEAR SCREEN
                  ;
                  ; ********************************************
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
```

```
                ;       CLEAR SCREEN, PRESERVING THE ZERO PAGE.
                ;
                ;
                ;
                ;
1100 20C411 CLR.TV JSR TVPUSH     SAVE ZERO PAGE BYTES THAT
                ;                 WILL BE CHANGED.
1103 202B11        JSR TVHOME     SET SCREEN LOCATION TO UPPER
                ;                 LEFT CORNER OF THE SCREEN.
1106 AE0310        LDX TVCOLS     LOAD X,Y REGISTERS WITH
1109 AC0410        LDY TVROWS     X,Y DIMENSIONS OF SCREEN.
110C 201311        JSR CLR.XY     CLEAR X COLUMNS, Y ROWS
                ;                 FROM CURRENT SCREEN LOCATION.
110F 20D311        JSR TV.POP     RESTORE ZERO PAGE BYTES THAT
                ;                 WERE CHANGED.
1112 60            RTS            RETURN TO CALLER, WITH ZERO
                ;                 PAGE PRESERVED.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;         CLEAR PORTION OF SCREEN
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;                 CLEAR X COLUMNS, Y ROWS
                ;                 FROM CURRENT SCREEN LOCATION.
                ;                 MOVES TV.PTR DOWN BY Y ROWS.
                ;
                ;
                ;
1113 8E2A11 CLR.XY STX COLS       SET THE NUMBER OF COLUMNS
                ;                 TO BE CLEARED.
1116 98            TYA
1117 AA            TAX            NOW X HOLDS NUMBER OF ROWS
                ;                 TO BE CLEARED.
                ;
1118 AD0610 CLRROW LDA BLANK      WE'LL CLEAR THEM BY
                ;                 WRITING BLANKS TO THE
                ;                 SCREEN.
111B AC2A11        LDY COLS       LOAD Y WITH NUMBER OF
                ;                 COLUMNS TO BE CLEARED.
111E 9100   CLRPOS STA (TV.PTR),Y CLEAR A POSITION BY
                ;                 WRITING A BLANK INTO IT.
                ;
1120 88            DEY            ADJUST INDEX FOR NEXT
```

```
                     ;                          POSITION ON THE ROW.
                     ;
     1121 10FB              BPL CLRPOS          IF NOT DONE WITH ROW,
                     ;                          CLEAR NEXT POSITION...
                     ;
     1123 207611            JSR TVDOWN          IF DONE WITH ROW, MOVE
                     ;                          CURRENT SCREEN LOCATION
                     ;                          DOWN BY ONE ROW.
                     ;
     1126 CA                DEX                 DONE LAST ROW YET?
     1127 10EF              BPL CLRROW          IF NOT, CLEAR NEXT ROW...
     1129 60                RTS                 IF SO, RETURN TO CALLER.
                     ;
     112A 00         COLS   .BYTE 0             DATA CELL: HOLDS NUMBER OF
                     ;                          COLUMNS TO BE CLEARED.
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ; ******************************************
                     ;
                     ;          TVHOME
                     ;
                     ; ******************************************
                     ;
                     ;
                     ;
                     ;
                     ;
     112B A200       TVHOME LDX #0              SET TV.PTR TO UPPER LEFT
     112D A000              LDY #0              CORNER OF SCREEN, BY
                     ;                          ZEROING X AND Y AND THEN
     112F 18                CLC                 GOING TO X,Y COORDINATES:
     1130 900A              BCC TVTOXY
                     ;
                     ;
                     ;
                     ;
                     ; ******************************************
                     ;
                     ;          CENTER
                     ;
                     ; ******************************************
                     ;
                     ;
                     ;
                     ;
                     ;                          SET TV.PTR TO SCREEN'S
                     ;                          CENTER:
                     ;
                     ;
                     ;
```

```
                        ;
1132 AD0410 CENTER LDA TVROWS     LOAD A WITH TOTAL ROWS.
1135 4A            LSR A          DIVIDE IT BY TWO.
1136 A8            TAY            Y NOW HOLDS THE NUMBER OF
                        ;         THE SCREEN'S CENTRAL ROW.
                        ;
1137 AD0310        LDA TVCOLS     LOAD A WITH TOTAL COLUMNS.
113A 4A            LSR A          DIVIDE IT BY TWO.
113B AA            TAX            X NOW HOLDS THE NUMBER OF
                        ;         THE SCREEN'S CENTRAL COLUMN.
                        ;
                        ;
                        ;         X AND Y REGISTERS NOW HOLD
                        ;         X,Y COORDINATES OF CENTER
                        ;         OF SCREEN.
                        ;
                        ;         SO NOW LET'S SET THE SCREEN
                        ;         LOCATION TO THOSE X,Y
                        ;         COORDINATES:
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ********************************************
                        ;
                        ;           TVTOXY
                        ;
                        ; ********************************************
                        ;
                        ;
                        ;
                        ;
                        ;
113C 38     TVTOXY SEC            SET CURRENT SCREEN LOCATION
                        ;         TO COORDINATES GIVEN BY
                        ;         THE X AND Y REGISTERS.
                        ;
113D EC0310        CPX TVCOLS     IS X OUT OF RANGE?
1140 9003          BCC X.OK       IF NOT, LEAVE IT ALONE.
                        ;         IF X IS OUT OF RANGE, GIVE
1142 AE0310        LDX TVCOLS     IT ITS HIGHEST LEGAL VALUE.
                        ;         NOW X IS LEGAL.
                        ;
1145 38     X.OK   SEC            IS Y OUT OF RANGE?
1146 CC0410        CPY TVROWS
1149 9003          BCC Y.OK       IF NOT, LEAVE IT ALONE.
                        ;
                        ;         IF Y IS OUT OF RANGE, GIVE
114B AC0410        LDY TVROWS     Y ITS HIGHEST LEGAL VALUE.
                        ;         NOW Y IS LEGAL.
                        ;
                        ;
114E AD0010 Y.OK   LDA HOME       SET TV.PTR = LOWEST SCREEN
```

```
2910 1151 8500          STA TV.PTR     ADDRESS.
2920 1153 AD0110        LDA HOME+1
2930 1156 8501          STA TV.PTR+1
2940               ;
2950 1158 08            PHP            SAVE CALLER'S DECIMAL FLAG.
2960 1159 D8            CLD            CLEAR DECIMAL FOR BINARY
2970               ;                   ADDITION.
2980               ;
2990 115A 8A            TXA            ADD X TO TV.PTR
3000 115B 18            CLC
3010 115C 6500          ADC TV.PTR
3020 115E 9003          BCC COLSET
3030 1160 E601          INC TV.PTR+1
3040 1162 18            CLC
3050               ;
3060               ;
3070 1163 C000   COLSET CPY #0         ADD Y*ROWINC TO TV.PTR:
3080 1165 F00B          BEQ TV.SET
3090 1167 18     ADDROW CLC
3100 1168 6D0210        ADC ROWINC
3110 116B 9002          BCC *+4
3120 116D E601          INC TV.PTR+1
3130 116F 88            DEY
3140 1170 D0F5          BNE ADDROW
3150               ;
3160               ;
3170 1172 8500   TV.SET STA TV.PTR
3180 1174 28            PLP            RESTORE CALLER'S DECIMAL FLAG
3190 1175 60            RTS            RETURN TO CALLER
3200               ;
3210               ;
3220               ;
3230               ;
3240               ;
3250               ;
3260               ;
3270               ;
3280               ;
3290               ;
3300               ; ********************************************
3310               ;
3320               ;         TVDOWN, TVSKIP, and TVPLUS
3330               ;
3340               ; ********************************************
3350               ;
3360               ;
3370               ;
3380               ;
3390               ;
3400 1176 AD0210 TVDOWN LDA ROWINC     MOVE TV.PTR DOWN BY ONE ROW.
3410 1179 18            CLC
3420 117A 9005          BCC TVPLUS
3430               ;
3440 117C 209B11 VUCHAR JSR TV.PUT     PUT CHARACTER ON SCREEN
3450               ;                   AND THEN
3460               ;
3470 117F A901   TVSKIP LDA #1         SKIP ONE SCREEN LOCATION
3480               ;                   BY INCREMENTING TV.PTR
```

```
                        ;
                        ;
1181 08        TVPLUS PHP              TVPLUS ADDS ACCUMULATOR
1182 D8               CLD              TO TV.PTR, KEEPING TV.PTR
1183 18               CLC              WITHIN SCREEN MEMORY.
1184 6500             ADC TV.PTR
1186 9002             BCC *+4
1188 E601             INC TV.PTR+1
118A 8500             STA TV.PTR
118C 38               SEC              IS CURRENT SCREEN LOCATION
118D AD0510           LDA HIPAGE       OUTSIDE OF SCREEN MEMORY?
1190 C501             CMP TV.PTR+1
1192 B005             BCS TV.OK
                        ;
1194 AD0110           LDA HOME+1       IF SO, WRAP AROUND FROM
1197 8501             STA TV.PTR+1     BOTTOM TO TOP OF SCREEN.
                        ;
1199 28        TV.OK  PLP              RESTORE ORIGINAL DECIMAL
119A 60               RTS              FLAG AND RETURN TO CALLER.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ********************************************
                        ;
                        ;          TV.PUT
                        ;
                        ; ********************************************
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
119B 201110    TV.PUT JSR FIXCHR       CONVERT ASCII CHARACTER
                        ;              TO YOUR SYSTEM'S DISPLAY
                        ;              CODE.
                        ;
119E A000             LDY #0           PUT CHARACTER AT CURRENT
11A0 9100             STA (TV.PTR),Y   SCREEN LOCATION.
11A2 60               RTS              THEN RETURN.
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ;
                        ; ********************************************
                        ;
```

```
                ;          DISPLAY A BYTE IN HEX FORMAT
                ;
                ; *********************************************
                ;
                ;
                ;
                ;
                ;
11A3 48         VUBYTE PHA             SAVE BYTE TO BE DISPLAYED.
11A4 4A                LSR A           MOVE 4 MOST SIGNIFICANT
11A5 4A                LSR A           BITS INTO POSITIONS
11A6 4A                LSR A           FORMERLY OCCUPIED BY 4
11A7 4A                LSR A           LEAST SIGNIFICANT BITS.
                ;
11A8 20B611            JSR ASCII       DETERMINE ASCII CHAR FOR
                ;                      HEX DIGIT IN A'S 4 LSB.
                ;
11AB 207C11            JSR VUCHAR      DISPLAY THAT ASCII CHAR ON
                ;                      SCREN AND ADVANCE TO NEXT
                ;                      SCREEN LOCATION.
                ;
11AE 68                PLA             RESTORE ORIGINAL BYTE TO A.
11AF 20B611            JSR ASCII       DETERMINE ASCII CHAR FOR
                ;                      A'S 4 LSB.
                ;
11B2 207C11            JSR VUCHAR      STORE THIS ASCII CHAR JUST
                ;                      TO THE RIGHT OF THE OTHER
                ;                      ASCII CHAR, AND ADVANCE TO
                ;                      NEXT SCREEN POSITION.
                ;
                ;
11B5 60                RTS             RETURN TO CALLER.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; *********************************************
                ;
                ;          HEX-TO-ASCII
                ;
                ; *********************************************
                ;
                ;
                ;
                ;
                ;
11B6 08         ASCII  PHP             THIS ROUTINE RETURNS ASCII
11B7 D8                CLD             FOR 4 LSB IN ACCUMULATOR.
11B8 290F              AND #$0F        CLEAR HIGH 4 BITS IN A.
11BA C90A              CMP #$0A        IS ACCUMULATOR GREATER
                ;                      THAN 9?
11BC 3002              BMI DECIML      IF NOT, IT MUST BE 0-9.
```

```
                ;
11BE 6906             ADC #6          IF SO, IT MUST BE A-F.
                ;                     ADD 36 HEX TO CONVERT IT.
                ;                     TO CORRESPONDING ASCII CHAR.
11C0 6930       DECIML ADC #$30       IF A IS 0-9, ADD 30 HEX
                ;                     TO CONVERT IT TO
                ;                     CORRESPONDING ASCII CHAR.
                ;
11C2 28               PLP             RESTORE ORIGINAL DECIMAL
                ;                     FLAG, AND
11C3 60               RTS             RETURN TO CALLER
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;          TVPUSH
                ;
                ; ********************************************
                ;
                ;
                ;
                ;                     SAVE CURRENT SCREEN LOCATION
                ;                     ON STACK, FOR CALLER.
                ;
                ;
                ;
                ;
                ;
11C4 68         TVPUSH PLA            PULL RETURN ADDRESS FROM
11C5 AA               TAX             STACK AND SAVE IT IN X AND
11C6 68               PLA             Y REGISTERS.
11C7 A8               TAY
                ;
                ;
11C8 A501             LDA TV.PTR+1    GET TV.PTR AND
11CA 48               PHA
11CB A500             LDA TV.PTR      PUSH IT ONTO THE STACK.
11CD 48               PHA
                ;
                ;
11CE 98               TYA             PLACE RETURN ADDRESS
11CF 48               PHA
11D0 8A               TXA             BACK ON STACK.
11D1 48               PHA
                ;
                ;
11D2 60               RTS             THEN RETURN TO CALLER.
```

```
                    ;                        CALLER WILL FIND TV.PTR ON
                    ;                        STACK, LOW BYTE ON TOP.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; ****************************************
                    ;
                    ;      TV.POP
                    ;
                    ; ****************************************
                    ;
                    ;
                    ;
                    ;                        RESTORE SCREEN LOCATION
                    ;                        PREVIOUSLY SAVED ON STACK.
                    ;
                    ;
                    ;
11D3 68            TV.POP PLA                PULL RETURN ADDRESS FROM
11D4 AA                   TAX                STACK, SAVING IT IN X...
11D5 68                   PLA
11D6 A8                   TAY                ...AND IN Y
                    ;
                    ;
11D7 68                   PLA                RESTORE...
11D8 8500                 STA TV.PTR         ...TV.PTR
11DA 68                   PLA                   ...FROM
11DB 8501                 STA TV.PTR+1              ...STACK.
                    ;
                    ;
11DD 98                   TYA                PLACE RETURN ADDRESS
11DE 48                   PHA                BACK ...
11DF 8A                   TXA
11E0 48                   PHA                ...ON STACK.
                    ;
                    ;
11E1 60                   RTS                RETURN TO CALLER.
```

Appendix C2:

Visible Monitor (Top Level and Display Subroutines)

```
10              ;     APPENDIX C2: ASSEMBLER LISTING OF
20              ;                THE VISIBLE MONITOR
30              ;
40              ;          TOP LEVEL AND DISPLAY SUBROUTINES
50              ;
60              ;
70              ;
80              ;
90              ;
100             ;       SEE CHAPTER 6 OF BEYOND GAMES: SYSTEMS
110             ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
120             ;
130             ;                   BY KEN SKIER
140             ;
150             ;
160             ;
170             ;
180             ;
190             ;
200             ;
210             ;
220             ;
230             ;
240             ;
250             ;
260             ;
270             ;
280             ;
290             ; ********************************************
300             ;
310             ;          EQUATES
320             ;
330             ; ******************************************
340             ;
350             ;
360             ;
370             ;
380 0000=             TV.PTR = 0
390             ;
400 0002=             GETPTR = 2
410             ;
420             ;
430 1000=             PARAMS = $1000 ADDRESS OF SYSTEM DATA
440             ;                    BLOCK.
450             ;
460             ;
470 1007=             ARROW = PARAMS+7
480             ;                    THIS DATA BYTE HOLDS YOUR
490             ;                    SYSTEM'S CHARACTER CODE
500             ;                    FOR AN UP-ARROW.
510             ;
520 1008=             ROMKEY = PARAMS+8
530             ;                    ROMKEY IS A POINTER TO
540             ;                    YOUR SYSTEM'S SUBROUTINE
550             ;                    TO GET AN ASCII CHARACTER
560             ;                    FROM THE KEYBOARD.
570             ;
580 0020=             SPACE = $20
```

```
590                   ;
600 007F=                    RUBOUT = $7F
610                   ;
620 000D=                    CR = $0D        ASCII FOR CARRIAGE RETURN.
630                   ;
640                   ;
650                   ;
660                   ;
670                   ;
680                   ;
690                   ;
700                   ;
710                   ;
720                   ;
730                   ;
740                   ;
750                   ; *******************************************
760                   ;
770                   ;           REQUIRED SUBROUTINES
780                   ;
790                   ; *******************************************
800                   ;
810                   ;
820                   ;
830 1100=                    TVSUBS = $1100
840 1100=                    CLR.TV = TVSUBS
850 1113=                    CLR.XY = TVSUBS+$13
860 112B=                    TVHOME = TVSUBS+$2B
870 113C=                    TVTOXY = TVSUBS+$3C
880 1176=                    TVDOWN = TVSUBS+$76
890 117C=                    VUCHAR = TVSUBS+$7C
900 117F=                    TVSKIP = TVSUBS+$7F
910 1181=                    TVPLUS = TVSUBS+$81
920 11A3=                    VUBYTE = TVSUBS+$A3
930 11B6=                    ASCII  = TVSUBS+$B6
940 11C4=                    TVPUSH = TVSUBS+$C4
950 11D3=                    TV.POP = TVSUBS+$D3
960                   ;
970                   ;
980                   ;
990 1200                     * = $1200
1000                  ;
1010                  ;
1020                  ;
1030 12E3=                   UPDATE = *+$E3
1040                  ;
1050                  ;
1060                  ;
1070                  ;
1080                  ;
1090                  ;
1100                  ;
1110                  ;
1120                  ;
1130                  ;
1140                  ;
1150                  ;
1160                  ;
```

```
                 ; *********************************************
                 ;
                 ;          USER-MODIFIABLE DATA
                 ;
                 ; *********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
1200 00         FIELD  .BYTE 0        NUMBER OF CURRENT FIELD.
                 ;                    (MUST BE 0-6.)
                 ;
1201 00         REG.A  .BYTE 0        IMAGE OF ACCUMULATOR.
                 ;
1202 00         REG.X  .BYTE 0        IMAGE OF X-REGISTER.
                 ;
1203 00         REG.Y  .BYTE 0        IMAGE OF Y-REGISTER.
                 ;
1204 00         REG.P  .BYTE 0        IMAGE OF PROCESSOR STATUS
                 ;                    REGISTER.
                 ;
1201=                  REGS = REG.A
                 ;
1205 0000       SELECT .WORD 0        POINTER TO CURRENTLY-
                 ;                    SELECTED ADDRESS.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *********************************************
                 ;
                 ;          THE VISIBLE MONITOR
                 ;
                 ; *********************************************
                 ;
                 ;
                 ;
                 ;
1207 08         VISMON PHP            SAVE CALLER'S STATUS FLAGS.
1208 D8                CLD            CLEAR DECIMAL MODE, SINCE
                 ;                    ARITHMETIC OPERATIONS IN THIS
                 ;                    BOOK ARE ALWAYS BINARY.
                 ;
1209 201212            JSR DSPLAY     PUT MONITOR DISPLAY ON
                 ;                    SCREEN.
                 ;
120C 20E312            JSR UPDATE     GET USER REQUEST AND
                 ;                    HANDLE IT.
120F 18                CLC
1210 90F6              BCC VISMON+1   LOOP BACK TO DISPLAY...
                 ;
                 ;
                 ;
```

```
1750                 ;
1760                 ;
1770                 ;
1780                 ;
1790                 ;
1800                 ;
1810                 ; *******************************************
1820                 ;
1830                 ;          MONITOR-DISPLAY
1840                 ;
1850                 ; *******************************************
1860                 ;
1870                 ;
1880                 ;
1890                 ;
1900                 ;
1910 1212 20C411 DSPLAY JSR TVPUSH     SAVE ZERO PAGE BYTES THAT
1920                 ;                 WILL BE MODIFIED.
1930                 ;
1940 1215 202512        JSR CLRMON     CLEAR A PORTION OF SCREEN.
1950 1218 203412        JSR LINE.1     DISPLAY LABEL LINE.
1960 121B 205C12        JSR LINE.2     DISPLAY DATA LINE.
1970 121E 20AF12        JSR LINE.3     DISPLAY ARROW LINE.
1980                 ;
1990 1221 20D311        JSR TV.POP     RESTORE ZERO PAGE BYTES
2000                 ;                 THAT WERE SAVED ABOVE.
2010                 ;
2020 1224 60            RTS            RETURN TO CALLER.
2030                 ;
2040                 ;
2050                 ;
2060                 ;
2070                 ;
2080                 ;
2090                 ;
2100                 ;
2110                 ;
2120                 ;
2130                 ; *******************************************
2140                 ;
2150                 ;          CLEAR PORTION OF SCREEN
2160                 ;
2170                 ; *******************************************
2180                 ;
2190                 ;
2200                 ;
2210                 ;
2220                 ;
2230 1225 A202   CLRMON LDX #2         SET TV.PTR TO COLUMN 2,
2240 1227 A002          LDY #2         ROW 2.
2250 1229 203C11        JSR TVTOXY
2260                 ;
2270 122C A219          LDX #25        LOAD X WITH NUMBER OF
2280                 ;                 COLUMNS (25) TO BE CLEARED.
2290                 ;
2300 122E A003          LDY #3         LOAD Y WITH NUMBER OF
2310                 ;                 ROWS (3) TO BE CLEARED.
2320                 ;
```

```
2330 1230 201311        JSR CLR.XY     CLEAR X COLUMNS, Y ROWS.
2340                ;
2350 1233 60            RTS            RETURN TO CALLER.
2360                ;
2370                ;
2380                ;
2390                ;
2400                ;
2410                ;
2420                ;
2430                ;
2440                ;
2450                ;
2460                ; ********************************************
2470                ;
2480                ;        DISPLAY LABEL LINE
2490                ;
2500                ; ********************************************
2510                ;
2520                ;
2530                ;
2540                ;
2550                ;
2560 1234 A20D   LINE.1 LDX #13        X-COORDINATE OF LABEL "A".
2570 1236 A002          LDY #2         Y-COORDINATE OF LABEL "A".
2580 1238 203C11        JSR TVTOXY     SET TV.PTR TO POINT TO
2590                ;                  SCREEN LOCATION OF LABEL "A
2600                ;
2610 123B A000          LDY #0         PUT LABELS ON SCREEN:
2620 123D 8C5112        STY LBLCOL     INITIALIZE LABEL COLUMN
2630                ;                  COUNTER.
2640                ;
2650 1240 B95212 LBLOOP LDA LABELS,Y   GET A CHARACTER AND
2660 1243 207C11        JSR VUCHAR     PUT IT ON THE SCREEN.
2670 1246 EE5112        INC LBLCOL     PREPARE FOR NEXT CHARACTER.
2680 1249 AC5112        LDY LBLCOL     DONE LAST CHARACTER?
2690 124C C00A          CPY #10
2700 124E D0F0          BNE LBLOOP     IF NOT, DO NEXT CHARACTER.
2710                ;
2720 1250 60            RTS            RETURN TO CALLER.
2730 1251 00     LBLCOL .BYTE 0        DATA CELL: HOLDS COLUMN
2740                ;                  OF CHARACTER TO BE COPIED.
2750                ;
2760                ;
2770                ;
2780                ;
2790 1252 41     LABELS .BYTE 'A  X  Y  P'
2790 1253 20
2790 1254 20
2790 1255 58
2790 1256 20
2790 1257 20
2790 1258 59
2790 1259 20
2790 125A 20
2790 125B 50
2800                ;
2810                ;
```

```
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ; *******************************************
                     ;
                     ;           DISPLAY DATA LINE
                     ;
                     ; *******************************************
                     ;
                     ;
                     ;
                     ;
                     ;
125C A202   LINE.2 LDX #2         LOAD X WITH STARTING
                     ;            COLUMN OF DATA LINE.
                     ;
125E A003          LDY #3         LOAD Y WITH ROW NUMBER
                     ;            OF DATA LINE.
                     ;
1260 203C11        JSR TVTOXY     SET TV.PTR TO POINT TO
                     ;            THE START OF THE DATA LINE.
                     ;
1263 AD0612        LDA SELECT+1   DISPLAY HIGH BYTE OF
1266 20A311        JSR VUBYTE     CURRENTLY-SELECTED ADDRESS.
1269 AD0512        LDA SELECT     DISPLAY LOW BYTE OF
126C 20A311        JSR VUBYTE     CURRENTLY-SELECTED ADDRESS.
                     ;
126F 207F11        JSR TVSKIP     SKIP ONE SPACE AFTER
                     ;            ADDRESS FIELD.
                     ;
1272 209412        JSR GET.SL     GET CURRENTLY-SELECTED
                     ;            BYTE.
                     ;
1275 48            PHA            SAVE IT.
                     ;
1276 20A311        JSR VUBYTE     DISPLAY IT, IN HEX FORMAT,
                     ;            IN FIELD 1.
                     ;
1279 207F11        JSR TVSKIP     SKIP ONE SPACE AFTER FIELD
                     ;            1.
                     ;
127C 68            PLA            RESTORE CURRENTLY-SELECTED
                     ;            BYTE TO ACCUMULATOR.
                     ;
127D 207C11        JSR VUCHAR     DISPLAY IT IN CHARACTER
                     ;            FORMAT, IN FIELD 2.
                     ;
1280 207F11        JSR TVSKIP     SKIP ONE SPACE AFTER FIELD 2.
                     ;
                     ;
                     ;            DISPLAY 6502 REGISTER
                     ;            IMAGES IN FIELDS 3-6:
                     ;
```

```
1283 A200          LDX #0          START WITH ACCUMULATOR
                 ;                 IMAGE.
1285 BD0112 VUREGS LDA REGS,X      LOOK UP THE REGISTER IMAGE.
1288 20A311        JSR VUBYTE      DISPLAY IT IN HEX FORMAT.
128B 207F11        JSR TVSKIP      SKIP ONE SPACE AFTER HEX
                 ;                 FIELD.
                 ;
128E E8            INX             GET READY FOR NEXT REGISTER...
128F E004          CPX #4          DONE FOUR REGISTERS YET?
1291 D0F2          BNE VUREGS      IF NOT, DO NEXT ONE...
                 ;
1293 60            RTS             IF ALL REGISTERS DISPLAYED,
                 ;                 RETURN.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *********************************************
                 ;
                 ;          GET SELECTED BYTE
                 ;
                 ; *********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
1294 A502   GET.SL LDA GETPTR      GET BYTE POINTED TO BY
1296 48            PHA             THE SELECT POINTER
1297 A603          LDX GETPTR+1    (PRESERVING THE ZERO PAGE).
                 ;
1299 AD0512        LDA SELECT
129C 8502          STA GETPTR
129E AD0612        LDA SELECT+1
12A1 8503          STA GETPTR+1
                 ;
12A3 A000          LDY #0
12A5 B102          LDA (GETPTR),Y
12A7 A8            TAY
12A8 68            PLA
12A9 8502          STA GETPTR
12AB 8603          STX GETPTR+1
12AD 98            TYA
12AE 60            RTS             RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
```

```
3980                 ;
3990                 ;
4000                 ;
4010                 ;
4020                 ; ********************************************
4030                 ;
4040                 ;           DISPLAY ARROW LINE
4050                 ;
4060                 ; ********************************************
4070                 ;
4080                 ;
4090                 ;
4100                 ;
4110                 ;
4120 12AF A202   LINE.3 LDX #2          LOAD X WITH STARTING COLUMN.
4130 12B1 A004          LDY #4          LOAD Y WITH ROW NUMBER.
4140 12B3 203C11        JSR TVTOXY      SET TV.PTR TO BEGINNING
4150                 ;                  OF ARROW LINE.
4160                 ;
4170 12B6 AC0012        LDY FIELD       LOOK UP CURRENT FIELD.
4180 12B9 38            SEC
4190 12BA C007          CPY #7
4200 12BC 9005          BCC FLD.OK
4210 12BE A000          LDY #0
4220 12C0 8C0012        STY FIELD
4230 12C3 B9CD12 FLD.OK LDA FIELDS,Y    LOOK UP COLUMN NUMBER FOR
4240                 ;                  CURRENT FIELD.
4250                 ;
4260 12C6 A8            TAY             USE THAT COLUMN NUMBER AS
4270                 ;                  AN INDEX INTO THE ROW.
4280                 ;
4290 12C7 AD0710        LDA ARROW       PLACE AN UP-ARROW IN
4300 12CA 9100          STA (TV.PTR),Y  COLUMN OF THE ARROW LINE.
4310 12CC 60            RTS             RETURN TO CALLER.
4320                 ;
4330                 ;
4340 12CD 03     FIELDS .BYTE 3,6,8     THIS DATA AREA SHOWS WHICH
4340 12CE 06
4340 12CF 08
4350 12D0 0B            .BYTE $0B,$0E   COLUMN SHOULD GET AN UP-
4350 12D1 0E
4360 12D2 11            .BYTE $11,$14   ARROW TO INDICATE ANY ONE
4360 12D3 14
4370                 ;                  OF FIELDS 0-6.  CHANGING
4380                 ;                  ONE OF THESE VALUES WILL
4390                 ;                  CAUSE THE UP-ARROW TO APPEAR
4400                 ;                  IN A DIFFERENT COLUMN WHEN
4410                 ;                  INDICATING A GIVEN FIELD.
4420                 ;
4430                 ;
```

Appendix C3:

Visible Monitor (Update Subroutine)

```
            ;     APPENDIX C3: ASSEMBLER LISTING OF
            ;                  THE VISIBLE MONITOR
            ;
            ;
            ;                   UPDATE SUBROUTINE
            ;
            ;
            ;
            ;     SEE CHAPTER 6 OF BEYOND GAMES: SYSTEMS
            ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
            ;
            ;                     BY KEN SKIER
            ;
            ;
            ;
            ;
            ;
            ;
            ;
            ;
            ;
            ;
            ; *******************************************
            ;
            ;          EQUATES
            ;
            ; *****************************************
            ;
            ;
            ;
            ;
            ;
            ;
            ;
0000=             TV.PTP = 0
            ;
0002=             GETPTR = 2
            ;
            ;
1000=             PARAMS = $1000 ADDRESS OF SYSTEM DATA
            ;                    BLOCK.
            ;
            ;
1007=             ARROW = PARAMS+7
            ;                    THIS DATA BYTE HOLDS YOUR
            ;                    SYSTEM'S CHARACTER CODE
            ;                    FOR AN UP-ARROW.
            ;
1008=             ROMKEY = PARAMS+8
            ;                    ROMKEY IS A POINTER TO
            ;                    YOUR SYSTEM'S SUBROUTINE
            ;                    TO GET AN ASCII CHARACTER
            ;                    FROM THE KEYBOARD.
            ;
1010=             DUMMY = PARAMS+$10
            ;                    DUMMY RETURNS WITHOUT DOING
            ;                    ANYTHING.
```

```
                 ;
                 ;
0020=                  SPACE = $20
                 ;
007F=                  RUBOUT = $7F
                 ;
000D=                  CR = $0D         ASCII FOR CARRIAGE RETURN.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;           REQUIRED SUBROUTINES
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
1100=                  TVSUBS = $1100
1100=                  CLR.TV = TVSUBS CLR.TV CLEARS THE SCREEN.
                 ;
                 ;
1200=                  VMSUBS = $1200 STARTING PAGE OF VISIBLE
                 ;                     MONITOR CODE.
                 ;
1294=                  GET.SL = VMSUBS+$94
                 ;                     GET.SL GETS THE CURRENTLY-
                 ;                     SELECTED BYTE.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;           USER-MODIFIABLE DATA
                 ;
                 ; ********************************************
                 ;
```

```
                   ;
                   ;
                   ;
                   ;
     1200                   * = VMSUBS
                   ;
                   ;
                   ;
                   ;
     1200 00       FIELD  .BYTE 0          NUMBER OF CURRENT FIELD.
                   ;                       (MUST BE 0-6.)
                   ;
     1201 00       REG.A  .BYTE 0          IMAGE OF ACCUMULATOR.
                   ;
     1202 00       REG.X  .BYTE 0          IMAGE OF X-REGISTER.
                   ;
     1203 00       REG.Y  .BYTE 0          IMAGE OF Y-REGISTER.
                   ;
     1204 00       REG.P  .BYTE 0          IMAGE OF PROCESSOR STATUS
                   ;                       REGISTER.
                   ;
     1201=                  REGS = REG.A
                   ;
     1205 0000     SELECT .WORD 0          POINTER TO CURRENTLY-
                   ;                       SELECTED ADDRESS.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ********************************************
                   ;
                   ;           KEYBOARD INPUT ROUTINE
                   ;
                   ; ********************************************
                   ;
                   ;
     12E0                   * = VMSUBS+$E0
                   ;
                   ;
     12E0 6C0810 GETKEY JMP (ROMKEY)    JSR GETKEY CALLS YOUR
                   ;                     SYSTEM'S KEYBOARD INPUT
                   ;                     ROUTINE INDIRECTLY.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
```

```
                      ;
                      ; ******************************************
                      ;
                      ;            MONITOR-UPDATE
                      ;
                      ; ******************************************
                      ;
                      ;
                      ;
                      ;
                      ;
12E3 20E012 UPDATE JSR GETKEY      GET A CHARACTER FROM THE
                      ;            KEYBOARD.
                      ;
12E6 C93E          CMP #'>         IS IT THE '>' KEY?
12E8 D010          BNE IF.LSR      IF NOT, PERFORM NEXT TEST.
                      ;
12EA EE0012 NEXT.F INC FIELD       IF SO, SELECT NEXT FIELD.
12ED AD0012        LDA FIELD
12F0 C907          CMP #7          IF ARROW WAS UNDER RIGHT-
12F2 D005          BNE UP.EX1      MOST FIELD, PLACE IT UNDER
12F4 A900          LDA #0          LEFT-MOST FIELD.
12F6 8D0012        STA FIELD
12F9 60     UP.EX1 RTS             THEN RETURN TO CALLER.
                      ;
                      ;
12FA C93C   IF.LSR CMP #'<         IS IT THE '<' KEY?
12FC D00B          BNE IF.SP       IF NOT, PERFORM NEXT TEST.
                      ;
12FE CE0012 PREV.F DEC FIELD       IF SO, SELECT PREVIOUS
1301 1005          BPL UP.EX2      FIELD: THE FIELD TO THE
1303 A906          LDA #6          LEFT OF THE CURRENT FIELD.
1305 8D0012        STA FIELD
1308 60     UP.EX2 RTS             THEN RETURN
                      ;
                      ;
1309 C920   IF.SP  CMP #SPACE      IS IT THE SPACE BAR?
130B D009          BNE IF.CR       IF NOT, PERFORM NEXT TEST.
                      ;
130D EE0512 INC.SL INC SELECT      IF SO, STEP FORWARD THROUGH
1310 D003          BNE *+5         MEMORY BY INCREMENTING
1312 EE0612        INC SELECT+1    THE POINTER THAT SELECTS
                      ;            THE ADDRESS TO BE DISPLAYED.
1315 60            RTS             THEN RETURN TO CALLER.
                      ;
                      ;
1316 C90D   IF.CR  CMP #CR         IS IT THE CARRIAGE RETURN?
1318 D00C          BNE IFCHAR      IF NOT, PERFORM NEXT TEST.
                      ;
131A AD0512 DEC.SL LDA SELECT      IF SO, STEP BACKWARD THROUGH
131D D003          BNE *+5         MEMORY BY DECREMENTING THE
131F CE0612        DEC SELECT+1    POINTER THAT SELECTS THE
1322 CE0512        DEC SELECT      ADDRESS TO BE DISPLAYED.
1325 60            RTS             THEN RETURN.
                      ;
                      ;
1326 AE0012 IFCHAR LDX FIELD       IS ARROW UNDER CHARACTER
1329 E002          CPX #2          FIELD (FIELD 2)?
```

```
132B D01B          BNE IF.GO        IF NOT, PERFORM NEXT TEST.
               ;                    IF SO,
132D A8     PUT.SL TAY              STORE THE
132E A500          LDA TV.PTR       CHARACTER IN THE CURRENTLY-
1330 48            PHA              SELECTED ADDRESS.
1331 A601          LDX TV.PTR+1     (PRESERVING THE ZERO PAGE.)
1333 AD0512        LDA SELECT
1336 8500          STA TV.PTR
1338 AD0612        LDA SELECT+1
133B 8501          STA TV.PTR+1
133D 98            TYA
133E A000          LDY #0
1340 9100          STA (TV.PTR),Y
1342 8601          STX TV.PTR+1
1344 68            PLA
1345 8500          STA TV.PTR
1347 60            RTS              THEN RETURN.
               ;
               ;
1348 C947   IF.GO  CMP #'G          IS IT 'G' FOR GO?
134A D023          BNE IF.HEX       IF NOT, PERFORM NEXT TEST.
               ;
134C AC0312 GO     LDY REG.Y        IF SO, LOAD REGISTERS
134F AE0212        LDX REG.X        FROM REGISTER IMAGES...
1352 AD0412        LDA REG.P
1355 48            PHA
1356 AD0112        LDA REG.A
1359 28            PLP
135A 206C13        JSR CALLIT       AND CALL SELECTED ADDRESS.
135D 08            PHP              WHEN THE SUBROUTINE RETURNS,
135E 8D0112        STA REG.A        SAVE REGISTER VALUES IN
1361 8E0212        STX REG.X        REGISTER IMAGES.
1364 8C0312        STY REG.Y
1367 68            PLA
1368 8D0412        STA REG.P
136B 60            RTS              THEN RETURN TO CALLER.
               ;
               ;
136C 6C0512 CALLIT JMP (SELECT)     JSR CALLIT CALLS THE
               ;                    CURRENTLY-SELECTED ADDRESS,
               ;                    INDIRECTLY.
               ;
               ;
136F 48     IF.HEX PHA              SAVE KEYBOARD CHARACTER.
1370 20D513        JSR BINARY       IS IT ASCII CHAR FOR 0-9 OR
               ;                    A-F?  IF SO, CONVERT TO BINARY.
               ;
               ;
1373 304B          BMI IF.CLR       IF KEYBOARD CHAR WAS N
               ;                    0-9 OR A-F, PERFORM NEXT
               ;                    TEST.
               ;
1375 A8            TAY              PULL KEYBOARD CHARACTER
1376 68            PLA              FROM STACK, WHILE SAVING
1377 98            TYA              BINARY EQUIVALENT IN A AND Y.
               ;
1378 AE0012        LDX FIELD        IS ARROW UNDER ADDRESS
137B D014          BNE NOTADR       FIELD (FIELD 0)?
```

```
                ;
1370 A203     ADRFLD LDX #3          SINCE ARROW IS UNDER ADDRESS
137F 18       ADLOOP CLC             FIELD, ROLL HEX DIGIT INTO
1380 0E0512          ASL SELECT      ADDRESS FIELD BY ROLLING IT
1383 2E0612          ROL SELECT+1    IT INTO THE POINTER THAT
1386 CA              DEX             SELECTS THE DISPLAYED
1387 10F6            BPL ADLOOP      ADDRESS.
1389 98              TYA
138A 0D0512          ORA SELECT
138D 8D0512          STA SELECT
1390 60              RTS             THEN RETURN.
                ;
                ;
1391 E001     NOTADR CPX #1          IS ARROW UNDER FIELD 1?
1393 D018            BNE REGFLD      IF NOT, IT MUST BE UNDER
                ;                    A REGISTER FIELD.
                ;
1395 290F     ROL.SL AND #$0F        ROLL 4 LSB IN A INTO
1397 48              PHA             CURRENTLY-SELECTED BYTE.
1398 209412          JSR GET.SL      GET THE CURRENTLY-SELECTED
139B 0A              ASL A           BYTE AND SHIFT LEFT 4 TIMES...
139C 0A              ASL A
139D 0A              ASL A
139E 0A              ASL A
139F 29F0            AND #$F0
13A1 8DAC13          STA TEMP
13A4 68              PLA
13A5 0DAC13          ORA TEMP
13A8 202D13          JSR PUT.SL      PUT IT IN CURRENTLY-SELECTED
13AB 60              RTS             ADDRESS AND RETURN.
                ;
13AC 00       TEMP   .BYTE 0
                ;
                ;
                ;
13AD CA       REGFLD DEX             THE ARROW MUST BE UNDER A
13AE CA              DEX             REGISTER IMAGE: FIELD 3,
13AF CA              DEX             4, 5, OR 6.
13B0 A003            LDY #3
                ;
13B2 18       RGLOOP CLC             ROLL HEX DIGIT INTO
13B3 1E0112          ASL REGS,X      APPROPRIATE REGISTER IMAGE.
13B6 88              DEY
13B7 10F9            BPL RGLOOP
13B9 1D0112          ORA REGS,X
13BC 9D0112          STA REGS,X
13BF 60              RTS
                ;
                ;
13C0 68       IF.CLR PLA             RESTORE KEYBOARD CHARACTER.
13C1 C97F            CMP #RUBOUT     IS IT RUBOUT? (IF YOUR
                ;                    SYSTEM DOESN'T HAVE A
                ;                    RUBOUT KEY, SUBSTITUTE THE
                ;                    CODE FOR THE KEY YOU'LL USE
                ;                    TO CLEAR THE SCREEN.)
                ;
13C3 D004            BNE NOTCLR      IF IT ISN'T THE 'CLEAR
                ;                    SCREEN' KEY, PERFORM NEXT
```

```
3480                 ;                    TEST.
3490                 ;
3500 13C5 200011          JSR CLR.TV      IF IT IS, THEN CLEAR THE
3510 13C8 60              RTS             SCREEN AND RETURN.
3520                 ;
3530                 ;
3540 13C9 C951   NOTCLR CMP #'Q           IS IT 'Q' FOR QUIT?
3550 13CB D004            BNE OTHER       IF NOT, PERFORM NEXT TEST.
3560                 ;
3570                 ;                    IT IS 'Q' FOR QUIT.  THE
3580                 ;                    USER WANTS TO RETURN TO THE
3590                 ;                    CALLER OF THE VISIBLE
3600                 ;                    MONITOR.  SO LET'S DO THAT:
3610 13CD 68              PLA             POP UPDATE'S RETURN ADDRESS.
3620 13CE 68              PLA
3630                 ;
3640 13CF 28              PLP             RESTORE INITIAL 6502 FLAGS.
3650                 ;                    VISMON'S RETURN ADDRESS IS
3660                 ;                    NOW ON THE STACK.
3670 13D0 60              RTS             SO RETURN TO CALLER OF
3680                 ;                    VISMON.  IN THIS WAY,
3690                 ;                    VISMON CAN BE USED BY ANY
3700                 ;                    CALLER TO GET AN ADDRESS
3710                 ;
3720                 ;                    FROM THE USER.
3730                 ;
3740 13D1 201010 OTHER    JSR DUMMY       REPLACE THIS CALL TO
3750                 ;                    DUMMY WITH A CALL TO ANY
3760                 ;                    SUBROUTINE THAT EXTENDS
3770                 ;                    FUNCTIONALITY OF THE
3780                 ;                    VISIBLE MONITOR.
3790 13D4 60              RTS             THEN RETURN.
3800                 ;
3810                 ;
3820                 ;
3830                 ;
3840                 ;
3850                 ;
3860                 ;
3870                 ;
3880                 ;
3890                 ;
3900                 ;
3910                 ;
3920                 ; ********************************************
3930                 ;
3940                 ;          ASCII TO BINARY
3950                 ;
3960                 ; ********************************************
3970                 ;
3980                 ;
3990                 ;
4000                 ;                    IF ACCUMULATOR HOLDS ASCII
4010                 ;                    0-9 OR A-F, THIS ROUTINE
4020                 ;                    RETURNS BINARY EQUIVALENT--
4030                 ;                    OTHERWISE, IT RETURNS $FF.
4040                 ;
4050                 ;
```

```
4060 13D5 38      BINARY SEC
4070 13D6 E930           SBC #$30
4080 13D8 900F           BCC BAD
4090 13DA C90A           CMP #$0A
4100 13DC 900E           BCC GOOD
4110 13DE E907           SBC #7
4120 13E0 C910           CMP #$10
4130 13E2 B005           BCS BAD
4140 13E4 38             SEC
4150 13E5 C90A           CMP #$0A
4160 13E7 B003           BCS GOOD
4170 13E9 A9FF    BAD    LDA #$FF
4180 13EB 60             RTS
4190               ;
4200 13EC A200    GOOD   LDX #0
4210 13EE 60             RTS
```

Appendix C4:

Print Utilities

```
              ;      APPENDIX C4: ASSEMBLER LISTING OF
              ;               PRINT UTILITIES
              ;
              ;
              ;
              ;
              ;      SEE CHAPTER 7 OF BEYOND GAMES: SYSTEMS
              ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ; ********************************************
              ;
              ;        CONSTANTS
              ;
              ; ********************************************
              ;
              ;
              ;
              ;
              ;
000D=             CR = $0D        CARRIAGE RETURN.
              ;
00FF=             ETX = $FF       THIS CHARACTER MUST
              ;                   TERMINATE ANY MESSAGE STRING.
              ;
000A=             LF = $0A        LINE FEED.
              ;
0000=             OFF = 0
              ;
00FF=             ON = $FF
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ; ********************************************
              ;
              ;        EXTERNAL ADDRESSES
              ;
              ; ********************************************
              ;
              ;
              ;
```

```
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
1000=                 PARAMS = $1000 ADDRESS OF SYSTEM DATA BLOCK.
                 ;
                 ;
                 ;
100C=                 ROMPRT = PARAMS+$0C
                 ;                       POINTER TO ROM ROUTINE THAT
                 ;                       SENDS CHAR TO SERIAL OUTPUT.
                 ;
                 ;
                 ;
100A=                 ROMTVT = PARAMS+$0A
                 ;                       POINTER TO ROM ROUTINE THAT
                 ;                       PRINTS A CHAR TO THE SCREEN.
                 ;
                 ;
                 ;
100E=                 USROUT = PARAMS+$0E
                 ;                       POINTER TO USER-WRITTEN
                 ;                       CHARACTER OUTPUT ROUTINE.
                 ;
                 ;
                 ;
                 ;
1100=                 TVSUBS = $1100
11B6=                 ASCII  = TVSUBS+$B6
                 ;
                 ;
                 ;
                 ;
1200=                 VMPAGE = $1200 VISIBLE MONITOR STARTING
                 ;                       PAGE
                 ;
1205=                 SELECT = VMPAGE+5
1294=                 GET.SL = VMPAGE+$94
130D=                 INC.SL = VMPAGE+$10D
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *********************************************
                 ;
                 ;        VARIABLES
                 ;
                 ; *********************************************
                 ;
                 ;
                 ;
                 ;
```

```
                    ;
     1400                   * = $1400
                    ;
     1400 00       PRINTR .BYTE OFF      PRINTER OUTPUT FLAG.
                    ;
     1401 FF       TVT    .BYTE ON       TVT OUTPUT FLAG.
                    ;
                    ;
     1402 00       USER   .BYTE OFF      OUTPUT FLAG FOR USER-
                    ;                    PROVIDED OUTPUT SUBROUTINE.
                    ;
     1403 00       CHAR   .BYTE 0        CHARACTER MOST RECENTLY
                    ;                    PRINTED BY PR.CHR.
                    ;                    CHAR=00 MEANS PR.CHR HAS
                    ;                    NEVER PRINTED A CHARACTER.
                    ;
                    ;
     1404 00       REPEAT .BYTE 0        THIS BYTE IS USED AS A
                    ;                    COUNTER BY SPACES, CHARS,
                    ;                    AND CR.LFS.
                    ;
                    ;
     1405 00       TEMP.X .BYTE 0        DATA CELL: USED BY PR.MSG.
                    ;
                    ;
     1406 0000     RETURN .WORD 0        THIS POINTER IS USED BY
                    ;                    PUSHSL AND POP.SL.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; *********************************************
                    ;
                    ;         DEVICE SELECT SUBROUTINES
                    ;
                    ; *********************************************
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
     1408 A9FF     TVT.ON LDA #ON        SELECT SCREEN FOR OUTPUT
     140A 8D0114          STA TVT        BY SETTING ITS DEVICE FLAG.
     140D 60              RTS
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
     140E A900     TVTOFF LDA #OFF       DE-SELECT SCREEN FOR
     1410 8D0114          STA TVT        OUTPUT BY CLEARING ITS
     1413 60              RTS            DEVICE FLAG.
```

```
                ;
                ;
                ;
                ;
                ;
1414 A9FF   PR.ON   LDA #ON       SELECT PRINTER FOR OUTPUT
1416 8D0014         STA PRINTR    BY SETTING ITS DEVICE FLAG.
1419 60             RTS
                ;
                ;
                ;
                ;
                ;
141A A900   PR.OFF  LDA #OFF      DE-SELECT PRINTER FOR OUTPUT
141C 8D0014         STA PRINTR    BY CLEARING ITS DEVICE FLAG.
141F 60             RTS
                ;
                ;
                ;
                ;
                ;
1420 A9FF   USR.ON  LDA #ON       SELECT USER-WRITTEN
1422 8D0214         STA USER      SUBROUTINE BY SETTING
1425 60             RTS           USER'S DEVICE FLAG.
                ;
                ;
                ;
                ;
                ;
1426 A900   USROFF  LDA #OFF      DE-SELECT USER-WRITTEN
1428 8D0214         STA USER      OUTPUT SUBROUTINE BY
142B 60             RTS           CLEARING ITS DEVICE FLAG.
                ;
                ;
                ;
                ;
                ;
142C 200814 ALL.ON  JSR TVT.ON    SELECT ALL OUTPUT DEVICES
142F 201414         JSR PR.ON     BY SELECTING EACH OUTPUT
1432 202014         JSR USR.ON    DEVICE INDIVIDUALLY.
1435 60             RTS
                ;
                ;
                ;
                ;
                ;
1436 200E14 ALLOFF  JSR TVTOFF    DE-SELECT ALL OUTPUT DEVICES
1439 201A14         JSR PR.OFF    BY DE-SELECTING EACH ONE
143C 202614         JSR USROFF    INDIVIDUALLY.
143F 60             RTS
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
```

```
                ;
                ;
                ; ******************************************
                ;
                ;         A GENERAL CHARACTER PRINT ROUTINE
                ;
                ; ******************************************
                ;
                ;
                ;
                ;
                ;
                ;     PRINT CHARACTER IN ACCUMULATOR
                ;
                ;  ON ALL CURRENTLY-SELECTED OUTPUT DEVICES.
                ;
                ;
                ;
1440 C900   PR.CHR CMP #0         TEST CHARACTER.
1442 F024          BEQ EXIT       IF IT'S A NULL, RETURN
                ;                 WITHOUT PRINTING IT.
1444 8D0314        STA CHAR       SAVE CHARACTER.
                ;
1447 AD0114        LDA TVT        IS SCREEN SELECTED?
144A F006          BEQ IF.PR      IF NOT, TEST NEXT DEVICE.
                ;
144C AD0314        LDA CHAR       IF SO, SEND CHARACTER
144F 206914        JSR SEND.1     INDIRECTLY TO SYSTEM'S
                ;                 TVT OUTPUT ROUTINE.
                ;
                ;
1452 AD0014 IF.PR  LDA PRINTR     IS PRINTER SELECTED?
1455 F006          BEQ IF.USR     IF NOT, TEST NEXT DEVICE.
                ;
1457 AD0314        LDA CHAR       IF SO, SEND CHARACTER
145A 206C14        JSR SEND.2     INDIRECTLY TO SYSTEM'S
                ;                 PRINTER DRIVER.
                ;
                ;
145D AD0214 IF.USR LDA USER       IS USER-WRITTEN OUTPUT
                ;                 SUBROUTINE SELECTED?
1460 F006          BEQ EXIT       IF NOT, RETURN.
                ;
1462 AD0314        LDA CHAR       IF SO, SEND CHARACTER
1465 206F14        JSR SEND.3     INDIRECTLY TO USER-WRITTEN
                ;                 SUBROUTINE.
                ;
1468 60     EXIT   RTS            RETURN TO CALLER.
                ;
                ;
                ;
                ;          VECTORED SUBROUTINE CALLS
                ;
                ;
                ;
1469 6C0A10 SEND.1 JMP (RONTVT)
                ;
146C 6C0C10 SEND.2 JMP (ROMPRT)
```

```
                 ;
146F 6C0E10 SEND.3 JMP (USROUT)
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;     SPECIALIZED CHARACTER OUTPUT ROUTINES
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;          PRINT A CARRIAGE RETURN-LINE FEED
                 ;
                 ;
1472 A90D   CR.LF  LDA #CR        SEND A CARRIAGE RETURN
1474 204014        JSR PR.CHR
1477 A90A          LDA #LF        AND A LINE-FEED TO ALL
1479 204014        JSR PR.CHR     CURRENTLY-SELECTED DEVICES.
147C 60            RTS            THEN RETURN.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;     PRINT A SPACE:
                 ;
                 ;
                 ;
147D A920   SPACE  LDA #$20       LOAD ACCUMULATOR WITH AN
147F 204014        JSR PR.CHR     ASCII SPACE AND PRINT IT.
1482 60            RTS            THEN RETURN.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;     PRINT BYTE
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
```

```
                 ;
                 ;
                 ;      PR.BYT OUTPUTS THE ACCUMULATOR, IN HEX,
                 ;    TO ALL CURRENTLY-SELECTED DEVICES.
                 ;
                 ;
                 ;
1483 48         PR.BYT PHA              SAVE BYTE.
1484 4A                LSR A            DETERMINE ASCII FOR 4 MSB...
1485 4A                LSR A
1486 4A                LSR A
1487 4A                LSR A
1488 20B611            JSR ASCII        ...IN THE BYTE.
148B 204014            JSR PR.CHR       PRINT THAT ASCII CHAR TO
                 ;                      CURRENT DEVICE(S).
148E 68                PLA              DETERMINE ASCII FOR 4 LSB
148F 20B611            JSR ASCII        IN THE ORIGINAL BYTE.
1492 204014            JSR PR.CHR       PRINT THAT CHARACTER.
1495 60                RTS              RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *********************************************
                 ;
                 ;      REPETITIVE CHARACTER OUTPUT
                 ;
                 ; *********************************************
                 ;
                 ;
                 ;
                 ;    PRINT X SPACES:
                 ;
                 ;
1496 A920       SPACES LDA #$20         LOAD A WITH ASCII SPACE.
                 ;
                 ;                      PRINT IT X TIMES:
                 ;
                 ;
                 ;
                 ;
                 ;    PRINT X CHARACTERS:
                 ;
                 ;
                 ;
1498 8E0414 CHARS  STX REPEAT           PRINT CHAR IN A X TIMES.
149B 48     RPLOOP PHA                  SAVE CHAR TO BE REPEATED.
149C AE0414        LDX REPEAT           REPEAT COUNTER TIMED OUT?
149F F00A          BEQ RPTEND           IF SO, EXIT.  IF NOT,
14A1 CE0414        DEC REPEAT           DECREMENT REPEAT COUNTER.
14A4 204014        JSR PR.CHR           PRINT CHARACTER.
                 ;
14A7 68            PLA                  RESTORE CHARACTER TO A.
14A8 18            CLC                  LOOP BACK TO PRINT IT
14A9 90F0          BCC RPLOOP           AGAIN IF NECESSARY.
```

```
                    ;
14AB 68     RPTEND PLA              CLEAN UP STACK AND
14AC 60            RTS              RETURN TO CALLER.
                    ;
                    ;
                    ;
                    ;      PRINT X NEWLINES
                    ;
                    ;
14AD 8E0414 CR.LFS STX REPEAT       INITIALIZE REPEAT COUNTER.
14B0 AE0414 CRLOOP LDX REPEAT       EXIT IF REPEAT COUNTER
14B3 F009          BEQ END.CR       HAS TIMED OUT.
14B5 CE0414        DEC REPEAT       DECREMENT REPEAT COUNTER.
14B8 207214        JSR CR.LF        PRINT A CARRIAGE RETURN
                    ;               AND A LINE FEED.
14BB 18            CLC              LOOP BACK TO SEE IF DONE
14BC 90F2          BCC CRLOOP       YET.
                    ;
14BE 60     END.CR RTS              RETURN TO CALLER.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; ********************************************
                    ;
                    ;        PRINT A MESSAGE
                    ;
                    ; ********************************************
                    ;
                    ;
                    ;
                    ;
                    ;       Xth POINTER IN ZERO PAGE
                    ;        POINTS TO THE MESSAGE.
                    ;
                    ;
14BF 8E0514 PR.MSG STX TEMP.X       SAVE X REGISTER, WHICH
                    ;               SPECIFIES MESSAGE POINTER.
                    ;
14C2 B501          LDA 1,X          SAVE MESSAGE POINTER.
14C4 48            PHA
14C5 B500          LDA 0,X
14C7 48            PHA
                    ;
14C8 AE0514 LOOP   LDX TEMP.X       RESTORE ORIGINAL X, SO IT
                    ;               SPECIFIES MESSAGE POINTER.
14CB A100          LDA (0,X)        GET NEXT CHARACTER FROM
14CD C9FF          CMP #ETX         MESSAGE.  IS MESSAGE OVER?
14CF F00C          BEQ MSGEND       IF SO, HANDLE END OF MESSAGE.
                    ;
14D1 F600          INC 0,X          IF NOT, INCREMENT POINTER.
14D3 D002          BNE NEXT         SO IT POINTS TO NEXT
14D5 F601          INC 1,X          CHARACTER IN MESSAGE.
```

```
4650 14D7 204014 NEXT   JSR PR.CHR     PRINT THE CHARACTER.
4660 14DA 18            CLC            LOOP BACK FOR NEXT
4670 14DB 90EB          BCC LOOP       CHARACTER...
4680                 ;
4690                 ;
4700 14DD 68     MSGEND PLA            RESTORE ORIGINAL MESSAGE
4710 14DE 9500          STA 0,X        POINTER.
4720 14E0 68            PLA
4730 14E1 9501          STA 1,X
4740 14E3 60            RTS            RETURN TO CALLER, WITH
4750                 ;                 MESSAGE POINTER PRESERVED.
4760                 ;
4770                 ;
4780                 ;
4790                 ;
4800                 ;
4810                 ;
4820                 ;
4830                 ;
4840                 ;
4850                 ; *********************************************
4860                 ;
4870                 ;       PRINT THE FOLLOWING TEXT
4880                 ;
4890                 ; *********************************************
4900                 ;
4910                 ;
4920                 ;
4930                 ;
4940                 ;
4950 14E4 68     PRINT: PLA            PULL RETURN ADDRESS FROM
4960 14E5 AA            TAX            STACK AND SAVE IT IN X AND
4970 14E6 68            PLA            Y REGISTERS.
4980 14E7 A8            TAY
4990                 ;
5000 14E8 201215        JSR PUSHSL     SAVE THE SELECT POINTER.
5010 14EB 8E0512        STX SELECT     SET SELECT=RETURN ADDRESS.
5020 14EE 8C0612        STY SELECT+1
5030                 ;
5040                 ;
5050 14F1 200D13        JSR INC.SL     ADVANCE SELECT TO STX.
5060                 ;
5070 14F4 200D13 NEXTCH JSR INC.SL     SELECT NEXT CHARACTER.
5080 14F7 209412        JSR GET.SL     GET IT.
5090 14FA C9FF          CMP #ETX       IS IT END OF MESSAGE?
5100 14FC F006          BEQ ENDIT      IF SO, RETURN.
5110 14FE 204014        JSR PR.CHR     IF NOT, PRINT CHARACTER.
5120 1501 18            CLC            LOOP BACK FOR NEXT
5130 1502 90F0          BCC NEXTCH     CHARACTER...
5140                 ;
5150                 ;
5160 1504 AE0512 ENDIT  LDX SELECT
5170 1507 AC0612        LDY SELECT+1
5180 150A 202B15        JSR POP.SL     RESTORE SELECT POINTER.
5190 150D 98            TYA            PUSH ADDRESS OF ETX ONTO
5200 150E 48            PHA
5210 150F 8A            TXA            ...THE STACK.
5220 1510 48            PHA
```

```
1511 60              RTS            RETURN (TO BYTE IMMEDIATELY
                                    FOLLOWING THE ETX.)
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; *******************************************
                ;
                ;        SAVE, RESTORE SELECT POINTER
                ;
                ; *******************************************
                ;
                ;
                ;
                ;
                ;
1512 68         PUSHSL PLA          PULL RETURN ADDRESS FROM
1513 8D0614           STA RETURN    STACK AND SAVE IT IN RETURN.
1516 68                PLA
1517 8D0714           STA RETURN+1
                ;
                ;
151A AD0612           LDA SELECT+1  PUSH SELECT POINTER ONTO
151D 48               PHA           THE STACK.
151E AD0512           LDA SELECT
1521 48               PHA
                ;
                ;
1522 AD0714           LDA RETURN+1  PUSH RETURN ADDRESS BACK
1525 48               PHA           ON THE STACK.
1526 AD0614           LDA RETURN
1529 48               PHA
                ;
                ;
152A 60               RTS           RETURN TO CALLER.  CALLER
                ;                   WILL FIND SELECT ON STACK.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
152B 68         POP.SL PLA          SAVE RETURN ADDRESS.
152C 8D0614           STA RETURN
152F 68               PLA
1530 8D0714           STA RETURN+1
                ;
                ;
1533 68               PLA           LOAD SELECT FROM STACK
1534 8D0512           STA SELECT
```

```
5810 1537 68            PLA
5820 1538 8D0612        STA SELECT+1
5830                ;
5840                ;
5850 153B AD0714        LDA RETURN+1    PLACE RETURN ADDRESS BACK
5860 153E 48            PHA             ON STACK.
5870 153F AD0614        LDA RETURN
5880 1542 48            PHA
5890                ;
5900                ;
5910 1543 60            RTS             RETURN TO CALLER.
5920                ;
```

Appendix C5:

Two Hexdump Tools

```
          ;      APPENDIX C5: ASSEMBLER LISTING OF
          ;               TWO HEXDUMP TOOLS
          ;
          ;
          ;
          ;      SEE CHAPTER 8 OF BEYOND GAMES: SYSTEMS
          ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
          ;
          ;
          ;                    BY KEN SKIER
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ; ********************************************
          ;
          ;        CONSTANTS
          ;
          ; ********************************************
          ;
          ;
          ;
          ;
          ;
000D=          CR = $0D        CARRIAGE RETURN.
          ;
000A=          LF = $0A        LINE FEED.
          ;
          ;
007F=          TEX = $7F       THIS CHARACTER MUST START
          ;                    ANY MESSAGE.
          ;
00FF=          ETX = $FF       THIS CHARACTER MUST END
          ;                    ANY MESSAGE.
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
```

```
                  ;
                  ; ******************************************
                  ;
                  ;         EXTERNAL ADDRESSES
                  ;
                  ; ******************************************
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
1100=                  TVSUBS=$1100   STARTING PAGE OF DISPLAY
                  ;                   CODE.
1100=                  CLR.TV=TVSUBS
11B6=                  ASCII =TVSUBS+$B6
                  ;
                  ;
1200=                  VMPAGE=$1200   STARTING PAGE OF VISIBLE
                  ;                   MONITOR CODE.
1205=                  SELECT=VMPAGE+5
1207=                  VISMON=VMPAGE+7
1294=                  GET.SL=VMPAGE+$94
130D=                  INC.SL=VMPAGE+$10D
                  ;
                  ;
1400=                  PRPAGE=$1400   STARTING PAGE OF PRINT
                  ;                   UTILITIES.
1408=                  TVT.ON=PRPAGE+8
140E=                  TVTOFF=PRPAGE+$0E
1414=                  PR.ON =PRPAGE+$14
141A=                  PR.OFF=PRPAGE+$1A
1440=                  PR.CHR=PRPAGE+$40
1472=                  CR.LF =PRPAGE+$72
147D=                  SPACE =PRPAGE+$7D
1496=                  SPACES=PRPAGE+$96
1483=                  PR.BYT=PRPAGE+$83
14E4=                  PRINT:=PRPAGE+$E4
1512=                  PUSHSL=PRPAGE+$112
152B=                  POP.SL=PRPAGE+$12B
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ******************************************
                  ;
                  ;         VARIABLES
```

```
                ;
                ; ******************************************
                ;
                ;
                ;
1550                    *=$1550
                ;
                ;
                ;
                ;
                ;
1550 00         COUNTR .BYTE 0          THIS BYTE COUNTS THE LINES
                ;                       DUMPED BY TVDUMP.
                ;
1551 04         NUMLNS .BYTE 4          NUMBER OF LINES TO BE
                ;                       DUMPED BY TVDUMP.
                ;   .
                ;
1552 0000       SA     .WORD 0          POINTER TO START OF MEMORY
                ;                       TO BE DUMPED BY PRDUMP.
1554 FFFF       EA     .WORD $FFFF      POINTER TO LAST BYTE TO
                ;                       BE DUMPED BY PRDUMP.
                ;
                ;
1556 00         COLUMN .BYTE 0          DATA CELL: USED BY PRLINE
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ******************************************
                ;
                ;         TVDUMP
                ;
                ; ******************************************
                ;
                ;
                ;
                ;
                ;
1557 200814 TVDUMP JSR TVT.ON           SELECT TVT AS OUTPUT DEVICE.
155A AD5115        LDA NUMLNS           SET COUNTR TO NUMBER OF
155D 8D5015        STA COUNTR           LINES TO BE DUMPED.
                ;
1560 AD0512        LDA SELECT           SET SELECT TO BEGINNING OF
1563 29F8          AND #$F8             A SCREEN LINE, BY ZEROING
1565 8D0512        STA SELECT           3 LSB IN SELECT.
                ;
1568 207214        JSR CR.LF            SKIP TWO LINES ON THE
156B 207214        JSR CR.LF            SCREEN.
                ;
156E 20A115 DUMPLN JSR PR.ADR           PRINT THE SELECTED ADDRESS.
                ;
1571 207214        JSR CR.LF            ADVANCE TO A NEW LINE ON
                ;                       SCREEN.  (NOT NEEDED ON
```

```
                   ;                         SYSTEMS WITH SCREENS MORE
                   ;                         THAN 27 COLUMNS WIDE.)
                   ;
                   ;
1574 207D14 DMPBYT JSR SPACE                 PRINT A SPACE TO THE SCREEN.
                   ;
1577 209A15        JSR DUMPSL                DUMP SELECTED BYTE.
                   ;
157A 200D13        JSR INC.SL                SELECT NEXT BYTE.
                   ;
157D AD0512        LDA SELECT                IS IT THE BEGINNING OF A
1580 2907          AND #07                   NEW SCREEN LINE (3 LSB=0?)
1582 D0F0          BNE DMPBYT                IF NOT, DUMP NEXT BYTE...
                   ;
                   ;
1584 207214        JSR CR.LF                 IF SO, ADVANCE TO A NEW LINE
                   ;                         ON THE SCREEN.
                   ;
1587 AD0512        LDA SELECT                DOES THIS ADDRESS MARK THE
158A 290F          AND #$0F                  BEGINNING OF A NEW HEX LINE?
                   ;                         (4 LSB = 0?)
                   ;
158C D003          BNE IFDONE
158E 207214        JSR CR.LF                 IF SO, ADVANCE TO A NEW
                   ;                         LINE ON SCREEN.
                   ;
1591 CE5015 IFDONE DEC COUNTR                DUMPED LAST LINE YET?
1594 D0D8          BNE DUMPLN                IF NOT, DUMP NEXT LINE.
                   ;
                   ;
1596 200E14        JSR TVTOFF                DE-SELECT TVT AS OUTPUT
                   ;                         DEVICE.
                   ;
1599 60            RTS                       RETURN TO CALLER.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ***********************************************
                   ;
                   ;          DUMP SELECTED BYTE
                   ;
                   ; ***********************************************
                   ;
                   ;
                   ;
                   ;
                   ;
159A 209412 DUMPSL JSR GET.SL                GET CURRENTLY-SELECTED BYTE
159D 208314        JSR PR.BYT                AND PRINT IT IN HEX FORMAT.
15A0 60            RTS                       RETURN TO CALLER.
                   ;
                   ;
```

```
2320               ;
2330               ;
2340               ;
2350               ;
2360               ;
2370               ;
2380               ;
2390               ;
2400               ;
2410               ; ********************************************
2420               ;
2430               ;         PRINT SELECTED ADDRESS
2440               ;
2450               ; ********************************************
2460               ;
2470               ;
2480               ;
2490               ;
2500               ;
2510               ;
2520 15A1 AD0612 PR.ADR LDA SELECT+1   FIRST PRINT THE HIGH BYTE...
2530 15A4 208314        JSR PR.BYT
2540 15A7 AD0512        LDA SELECT     ...THEN PRINT THE LOW BYTE.
2550 15AA 208314        JSR PR.BYT
2560 15AD 60            RTS
2570               ;
2580               ;
2590               ;
2600               ;
2610               ;
2620               ;
2630               ;
2640               ;
2650               ;
2660               ; ********************************************
2670               ;
2680               ;       PRINTING HEXDUMP
2690               ;
2700               ; ********************************************
2710               ;
2720               ;
2730               ;
2740               ;
2750               ;
2760 15AE 20C915 PRDUMP JSR TITLE      DISPLAY THE TITLE
2770 15B1 20E915        JSR SETADS     LET USER SET START ADDRESS
2780               ;                   AND END ADDRESS OF MEMORY TO
2790               ;                   BE DUMPED.
2800               ;                   (SETADS RETURNS W/SELECT=EA.)
2810 15B4 20A017        JSR GOTOSA     SET SELECT=SA.
2820 15B7 201414        JSR PR.ON      SELECT PRINTER FOR OUTPUT.
2830               ;
2840 15BA 20EB16        JSR HEADER     OUTPUT HEXDUMP HEADER.
2850               ;
2860               ;
2870 15BD 204217 HXLOOP JSR PRLINE     DUMP ONE LINE.
2880 15C0 10FB          BPL HXLOOP     DUMPED LAST LINE? IF NOT,
2890               ;                   DUMP NEXT LINE.
```

```
                 ;
15C2 207214           JSR CR.LF      IF SO, GO TO A NEW LINE.
                 ;
15C5 201A14           JSR PR.OFF     DE-SELECT PRINTER FOR OUTPUT.
                 ;
15C8 60               RTS            RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;      PRINT THE HEXDUMP TITLE TO SCREEN
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
15C9 200011 TITLE     JSR CLR.TV     CLEAR THE SCREEN.
15CC 200814           JSR TVT.ON     SELECT SCREEN FOR OUTPUT.
15CF 20E414           JSR PRINT:     OUTPUT THE FOLLOWING TEXT:
15D2 7F               .BYTE TEX      TEXT STRING MUST START
                 ;                   WITH A START OF TEXT CHAR.
15D3 0D               .BYTE CR,'PRINTING HEXDUMP',CR,LF,LF
15D4 50
15D5 52
15D6 49
15D7 4E
15D8 54
15D9 49
15DA 4E
15DB 47
15DC 20
15DD 48
15DE 45
15DF 58
15E0 44
15E1 55
15E2 4D
15E3 50
15E4 0D
15E5 0A
15E6 0A
15E7 FF               .BYTE ETX      TEXT STRING MUST END WITH
                 ;                   AN END OF TEXT CHARACTER.
15E8 60               RTS            RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
```

```
3290                ;
3300                ;
3310                ;
3320                ;
3330                ;
3340                ; *********************************************
3350                ;
3360                ;     LET USER SET STARTING ADDRESS AND
3370                ;
3380                ;        END ADDRESS OF A BLOCK OF MEMORY:
3390                ;
3400                ; *********************************************
3410                ;
3420                ;
3430                ;
3440                ;
3450                ;
3460 15E9 200814 SETADS JSR TVT.ON     SELECT SCREEN FOR OUTPUT
3470 15EC 20E414        JSR PRINT:     PUT PROMPT ON SCREEN:
3480 15EF 7F            .BYTE TEX
3490 15F0 0D            .BYTE CR,LF,'SET STARTING ADDRESS '
3490 15F1 0A
3490 15F2 53
3490 15F3 45
3490 15F4 54
3490 15F5 20
3490 15F6 53
3490 15F7 54
3490 15F8 41
3490 15F9 52
3490 15FA 54
3490 15FB 49
3490 15FC 4E
3490 15FD 47
3490 15FE 20
3490 15FF 41
3490 1600 44
3490 1601 44
3490 1602 52
3490 1603 45
3490 1604 53
3490 1605 53
3490 1606 20
3500 1607 41            .BYTE 'AND PRESS "Q".'
3500 1608 4E
3500 1609 44
3500 160A 20
3500 160B 50
3500 160C 52
3500 160D 45
3500 160E 53
3500 160F 53
3500 1610 20
3500 1611 22
3500 1612 51
3500 1613 22
3500 1614 2E
3510 1615 FF            .BYTE ETX
```

```
3520 1616 200712        JSR VISMON     CALL VISIBLE MONITOR. SO
3530                ;                  USER CAN SELECT START ADDRESS
3540                ;                  OF THE BLOCK.
3550                ;
3560 1619 206716        JSR SAHERE     SET START ADDRESS (SA)=SELECT
3570                ;
3580                ;
3590                ;
3600                ;
3610                ;
3620                ;
3630                ;                  HAVING SET THE START ADDRESS,
3640                ;                  SA, LET'S SET THE END ADDRESS,
3650                ;                  EA.
3660                ;
3670                ;
3680                ;
3690                ;
3700                ;
3710 161C 200814 SET.EA JSR TVT.ON     SELECT SCREEN FOR OUTPUT.
3720 161F 20E414        JSR PRINT:     PUT PROMPT ON SCREEN:
3730 1622 7F            .BYTE TEX
3740 1623 0D            .BYTE CR,LF,'SET END ADDRESS '
3740 1624 0A
3740 1625 53
3740 1626 45
3740 1627 54
3740 1628 20
3740 1629 45
3740 162A 4E
3740 162B 44
3740 162C 20
3740 162D 41
3740 162E 44
3740 162F 44
3740 1630 52
3740 1631 45
3740 1632 53
3740 1633 53
3740 1634 20
3750 1635 41            .BYTE 'AND PRESS "Q".',ETX
3750 1636 4E
3750 1637 44
3750 1638 20
3750 1639 50
3750 163A 52
3750 163B 45
3750 163C 53
3750 163D 53
3750 163E 20
3750 163F 22
3750 1640 51
3750 1641 22
3750 1642 2E
3750 1643 FF
3760                ;
3770 1644 200712        JSR VISMON     LET USER SELECT END ADDRESS.
3780                ;
```

```
3790 1647 38            SEC             IF USER TRIED TO SET AN
3800 1648 AD0612        LDA SELECT+1    ADDRESS LESS THAN THE
3810 164B CD5315        CMP SA+1        STARTING ADDRESS,
3820 164E 9024          BCC TOOLOW      MAKE USER DO IT OVER.
3830 1650 D008          BNE EAHERE      IF SELECT>SA, SET EA=SELECT.
3840               ;                    THAT WILL MAKE EA>SA,
3850               ;
3860               ;
3870               ;
3880 1652 AD0512        LDA SELECT
3890 1655 CD5215        CMP SA
3900 1658 901A          BCC TOOLOW
3910               ;
3920               ;
3930               ;
3940               ;
3950 165A AD0612 EAHERE LDA SELECT+1    SET EA=SELECT.
3960 165D 8D5515        STA EA+1
3970 1660 AD0512        LDA SELECT
3980 1663 8D5415        STA EA
3990 1666 60            RTS             RETURN WITH EA SET BY CALLER
4000               ;                    (JSR EAHERE); EA SET BY USER
4010               ;                    (JSR SET.EA); OR SA AND EA
4020               ;                    SET BY USER (JSR SETADS).
4030               ;
4040 1667 AD0612 SAHERE LDA SELECT+1    SET SA=SELECT.
4050 166A 8D5315        STA SA+1
4060 166D AD0512        LDA SELECT
4070 1670 8D5215        STA SA
4080 1673 60            RTS             RETURN WITH SA=SELECT.
4090               ;
4100 1674 20E414 TOOLOW JSR PRINT:      SINCE USER SET ENDING
4110               ;                    ADDRESS TOO LOW, PUT A
4120               ;                    PROMPT ON THE SCREEN:
4130 1677 7F            .BYTE TEX
4140 1678 0D            .BYTE CR,LF,LF,LF,' ERROR!!! '
4140 1679 0A
4140 167A 0A
4140 167B 0A
4140 167C 20
4140 167D 45
4140 167E 52
4140 167F 52
4140 1680 4F
4140 1681 52
4140 1682 21
4140 1683 21
4140 1684 21
4140 1685 20
4150 1686 45            .BYTE 'END ADDRESS LESS THAN START ADDRESS,'
4150 1687 4E
4150 1688 44
4150 1689 20
4150 168A 41
4150 168B 44
4150 168C 44
4150 168D 52
4150 168E 45
```

```
4150 168F 53
4150 1690 53
4150 1691 20
4150 1692 4C
4150 1693 45
4150 1694 53
4150 1695 53
4150 1696 20
4150 1697 54
4150 1698 48
4150 1699 41
4150 169A 4E
4150 169B 20
4150 169C 53
4150 169D 54
4150 169E 41
4150 169F 52
4150 16A0 54
4150 16A1 20
4150 16A2 41
4150 16A3 44
4150 16A4 44
4150 16A5 52
4150 16A6 45
4150 16A7 53
4150 16A8 53
4150 16A9 2C
4160 16AA 20           .BYTE ' WHICH IS ',ETX
4160 16AB 57
4160 16AC 48
4160 16AD 49
4160 16AE 43
4160 16AF 48
4160 16B0 20
4160 16B1 49
4160 16B2 53
4160 16B3 20
4160 16B4 FF
4170 16B5 205B16       JSR PR.SA       PRINT START ADDRESS.
4180                ;
4190 16B8 4C1C16       JMP SET.EA      AND LET THE USER SET A
4200                ;                  NEW END ADDRESS.
4210                ;
4220                ;
4230                ;
4240                ;
4250                ;
4260                ;
4270                ;
4280                ;
4290                ;
4300                ; ***********************************************
4310                ;
4320                ;         PRINT START ADDRESS
4330                ;
4340                ; ***********************************************
4350                ;
4360                ;
```

```
4370                 ;
4380                 ;
4390 16BB A924   PR.SA  LDA #'$        PRINT A DOLLAR SIGN, TO
4400 16BD 204014        JSR PR.CHR     INDICATE HEXADECIMAL.
4410                 ;
4420 16C0 AD5315        LDA SA+1       PRINT HIGH BYTE OF START
4430 16C3 208314        JSR PR.BYT     ADDRESS.
4440                 ;
4450 16C6 AD5215        LDA SA         PRINT LOW BYTE OF START
4460 16C9 208314        JSR PR.BYT
4470 16CC 60            RTS            RETURN TO CALLER.
4480                 ;
4490                 ;
4500                 ;
4510                 ;
4520                 ;
4530                 ;
4540                 ;
4550                 ; ********************************************
4560                 ;
4570                 ;          PRINT END ADDRESS
4580                 ;
4590                 ; ********************************************
4600                 ;
4610                 ;
4620                 ;
4630                 ;
4640                 ;
4650 16CD A924   PR.EA  LDA #'$        PRINT A DOLLAR SIGN, TO
4660 16CF 204014        JSR PR.CHR     INDICATE HEXADECIMAL.
4670 16D2 AD5515        LDA EA+1       PRINT HIGH BYTE OF END
4680 16D5 208314        JSR PR.BYT     ADDRESS.
4690 16D8 AD5415        LDA EA         PRINT LOW BYTE OF END
4700 16DB 208314        JSR PR.BYT     ADDRESS.
4710 16DE 60            RTS            RETURN TO CALLER.
4720                 ;
4730                 ;
4740                 ;
4750                 ;
4760                 ;
4770                 ;
4780                 ;
4790                 ;
4800                 ;
4810                 ;
4820                 ; ********************************************
4830                 ;
4840                 ;          PRINT RANGE OF ADDRESSES
4850                 ;
4860                 ; ********************************************
4870                 ;
4880                 ;
4890                 ;
4900                 ;
4910                 ;
4920 16DF 20BB16 RANGE  JSR PR.SA      PRINT STARTING ADDRESS.
4930 16E2 A92D          LDA #'-        PRINT A HYPHEN.
4940 16E4 204014        JSR PR.CHR
```

```
4950 16E7 20CD16        JSR PR.EA       PRINT END ADDRESS.
4960 16EA 60            RTS             RETURN TO CALLER.
4970                ;
4980                ;
4990                ;
5000                ;
5010                ;
5020                ;
5030                ;
5040                ;
5050                ;
5060                ;
5070                ; ********************************************
5080                ;
5090                ;         PRINT HEADER
5100                ;
5110                ; ********************************************
5120                ;
5130                ;
5140                ;
5150                ;
5160                ;
5170 16EB 20E414 HEADER JSR PRINT:
5180 16EE 7F            .BYTE TEX
5190 16EF 0D            .BYTE CR,LF,LF,'DUMPING '
5190 16F0 0A
5190 16F1 0A
5190 16F2 44
5190 16F3 55
5190 16F4 4D
5190 16F5 50
5190 16F6 49
5190 16F7 4E
5190 16F8 47
5190 16F9 20
5200 16FA FF            .BYTE ETX
5210 16FB 20DF16        JSR RANGE
5220 16FE 207214        JSR CR.LF
5230 1701 20E414        JSR PRINT:
5240 1704 7F            .BYTE TEX,LF,LF
5240 1705 0A
5240 1706 0A
5250 1707 20            .BYTE '        0  1  2  3  4  5  6  7 '
5250 1708 20
5250 1709 20
5250 170A 20
5250 170B 20
5250 170C 20
5250 170D 20
5250 170E 20
5250 170F 30
5250 1710 20
5250 1711 20
5250 1712 31
5250 1713 20
5250 1714 20
5250 1715 32
5250 1716 20
```

```
5250 1717 20
5250 1718 33
5250 1719 20
5250 171A 20
5250 171B 34
5250 171C 20
5250 171D 20
5250 171E 35
5250 171F 20
5250 1720 20
5250 1721 36
5250 1722 20
5250 1723 20
5250 1724 37
5250 1725 20
5250 1726 20
5260 1727 38                 .BYTE '8  9  A  B  C  D  E  F'
5260 1728 20
5260 1729 20
5260 172A 39
5260 172B 20
5260 172C 20
5260 172D 41
5260 172E 20
5260 172F 20
5260 1730 42
5260 1731 20
5260 1732 20
5260 1733 43
5260 1734 20
5260 1735 20
5260 1736 44
5260 1737 20
5260 1738 20
5260 1739 45
5260 173A 20
5260 173B 20
5260 173C 46
5270 173D 0D                 .BYTE CR,LF,LF,ETX
5270 173E 0A
5270 173F 0A
5270 1740 FF
5280 1741 60                 RTS
5290                ;
5300                ;
5310                ;
5320                ;
5330                ;
5340                ;
5350                ;
5360                ;
5370                ;
5380                ;
5390                ; *******************************************
5400                ;
5410                ;         DUMP ONE LINE TO PRINTER
5420                ;
5430                ; *******************************************
```

```
                  ;
                  ;
                  ;
                  ;
                  ;
1742 207214 PRLINE JSR CR.LF
1745 AD0512        LDA SELECT    DETERMINE STARTING COLUMN.
1748 48            PHA           FOR THIS DUMP.
1749 290F          AND #$0F
174B 8D5615        STA COLUMN    NOW COLUMN HOLDS NUMBER OF
                  ;              HEX COLUMN IN WHICH WE DUMP
                  ;              THE FIRST BYTE.
174E 68            PLA           SET SELECT=BEGINNING OF A
174F 29F0          AND #$F0      HEX LINE.
1751 8D0512        STA SELECT
1754 20A115        JSR PR.ADR    PRINT LINE'S START ADDRESS.
1757 A203          LDX #3        SPACE 3 TIMES--TO THE
1759 209614        JSR SPACES    FIRST HEX COLUMN.
                  ;
                  ;
175C AD5615        LDA COLUMN    DO WE DUMP FROM THE FIRST
                  ;              HEX COLUMN?
175F F00D          BEQ COL.OK    IF SO, WERE AT THE CORRECT
                  ;              COLUMN NOW.
                  ;
1761 A203   LOOP   LDX #3        IF NOT, SPACE 3 TIMES FOR
1763 209614        JSR SPACES    EACH BYTE NOT DUMPED.
1766 200D13        JSR INC.SL
1769 CE5615        DEC COLUMN
176C D0F3          BNE LOOP
                  ;
176E 209A15 COL.OK JSR DUMPSL    DUMP SELECTED BYTE.
1771 207D14        JSR SPACE     SPACE ONCE.
1774 208317        JSR NEXTSL    SELECT NEXT BYTE
                  ;
1777 3009          BMI EXIT      MINUS MEANS WE'VE DUMPED
                  ;              THROUGH TO THE END ADDRESS.
                  ;
                  ;
1779 AD0512 NOT.EA LDA SELECT    DUMPED ENTIRE LINE?
177C 290F          AND #$0F      (4LSB OF SELECT=0?)
177E C900          CMP #0        IF SO, WE'VE DUMPED THE
                  ;              ENTIRE LINE.  IF NOT,
1780 D0EC          BNE COL.OK    SELECT NEXT BYTE AND DUMP IT.
1782 60     EXIT   RTS           RETURN MINUS IF EA DUMPED;
                  ;              RETURN PLUS IF EA NOT DUMPED.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ***********************************************
                  ;
```

```
6020                ;       SELECT NEXT BYTE (IF < END ADDRESS)
6030                ;
6040                ; ********************************************
6050                ;
6060                ;
6070                ;
6080                ;
6090                ;
6100 1783 38      NEXTSL SEC
6110 1784 AD0612         LDA SELECT+1    HIGH BYTE OF SELECT LESS
6120 1787 CD5515         CMP EA+1        THAN HIGH BYTE OF EA?
6130 178A 900B           BCC SL.OK       IF SO, SELECT<END ADDRESS.
6140 178C D00F           BNE NO.INC      IF SELECT>EA, DON'T
6150                ;                    INCREMENT SELECT.
6160                ;
6170 178E 38             SEC             SELECT IS IN SAME PAGE AS EA.
6180 178F AD0512         LDA SELECT
6190 1792 CD5415         CMP EA
6200 1795 B006           BCS NO.INC
6210                ;
6220 1797 200D13 SL.OK  JSR INC.SL       SINCE SELECT <= EA, WE MAY
6230                ;                    INCREMENT SELECT.
6240                ;
6250 179A A900           LDA #0          SET "INCREMENTED" RETURN
6260 179C 60             RTS             CODE AND RETURN.
6270                ;
6280 179D A9FF    NO.INC LDA #$FF        SET "NO INCREMENT" RETURN
6290 179F 60             RTS             CODE AND RETURN.
6300                ;
6310                ;
6320                ;
6330                ;
6340                ;
6350                ;
6360                ; ********************************************
6370                ;
6380                ;         SELECT START ADDRESS
6390                ;
6400                ; ********************************************
6410                ;
6420                ;
6430                ;
6440                ;
6450                ;
6460 17A0 AD5215 GOTOSA LDA SA           SET SELECT=SA.
6470 17A3 8D0512         STA SELECT
6480 17A6 AD5315         LDA SA+1
6490 17A9 8D0612         STA SELECT+1
6500 17AC 60             RTS             RETURN W?SELECT=SA.
```

Appendix C6:

Table-Driven Disassembler (Top Level and Utility Subroutines)

```
                  ;       APPENDIX C6: ASSEMBLER LISTING OF
                  ;          TABLE-DRIVEN DISASSEMBLER
                  ;
                  ;       TOP-LEVEL AND UTILITY SUBROUTINES
                  ;
                  ;
                  ;
                  ;
                  ;     SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM
                  ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                  ;
                  ;
                  ;                  BY KEN SKIER
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;         CONSTANTS
                  ;
                  ; ********************************************
                  ;
                  ;
                  ;
                  ;
                  ;
    000D=               CR = $0D        CARRIAGE RETURN.
                  ;
    000A=               LF = $0A        LINE FEED.
                  ;
                  ;
    007F=               TEX = $7F       THIS CHARACTER MUST START
                                        ANY MESSAGE.
                  ;
                  ;
    00FF=               ETX = $FF       THIS CHARACTER MUST END
                                        ANY MESSAGE.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;         EXTERNAL ADDRESSES
                  ;
                  ; ********************************************
```

```
590                    ;
600                    ;
610                    ;
620                    ;
630                    ;
640 1200=                 VMPAGE=$1200   STARTING PAGE OF VISIBLE
650                    ;                 MONITOR CODE.
660 1205=                 SELECT=VMPAGE+5
670 1207=                 VISMON=VMPAGE+7
680 1294=                 GET.SL=VMPAGE+$94
690 130D=                 INC.SL=VMPAGE+$10D
700 131A=                 DEC.SL=VMPAGE+$11A
710                    ;
720                    ;
730 1400=                 PRPAGE=$1400   STARTING PAGE OF PRINT
740                    ;                 UTILITIES.
750 1408=                 TVT.ON=PRPAGE+8
760 140E=                 TVTOFF=PRPAGE+$0E
770 1414=                 PR.ON =PRPAGE+$14
780 141A=                 PR.OFF=PRPAGE+$1A
790 1440=                 PR.CHR=PRPAGE+$40
800 1472=                 CR.LF =PRPAGE+$72
810 147D=                 SPACE =PRPAGE+$7D
820 1496=                 SPACES=PRPAGE+$96
830 1483=                 PR.BYT=PRPAGE+$83
840 14E4=                 PRINT:=PRPAGE+$E4
850 1512=                 PUSHSL=PRPAGE+$112
860 152B=                 POP.SL=PRPAGE+$12B
870                    ;
880                    ;
890 1500=                 HEX.PG=$1500   ADDRESS OF PAGE IN WHICH
900                    ;                 HEXDUMP CODE STARTS.
910                    ;
920 1552=                 SA=HEX.PG+$52
930 1554=                 EA=SA+2
940 159A=                 DUMPSL=HEX.PG+$9A
950 15A1=                 PR.ADR=HEX.PG+$A1
960 16DF=                 RANGE=HEX.PG+$1DF
970 15E9=                 SETADS=HEX.PG+$E9
980 1783=                 NEXTSL=HEX.PG+$283
990 17A0=                 GOTOSA=HEX.PG+$2A0
1000                   ;
1010                   ;
1020                   ;
1030                   ;
1040                   ;
1050                   ;       DISASSEMBLER TABLES:
1060                   ;
1070                   ;
1080                   ;
1090 1900=                DSPAGE=$1900   STARTING PAGE OF DISASSEMBLER
1100                   ;
1110 1B1B=                SUBS  =DSPAGE+$21B
1120 1B50=                MNAMES=DSPAGE+$250
1130 1C00=                MCODES=DSPAGE+$300
1140 1D00=                MODES =DSPAGE+$400
1150                   ;
1160                   ;
```

```
                 ;
                 ;
                 ; *******************************************
                 ;
                 ;         VARIABLES
                 ;
                 ; *******************************************
                 ;
                 ;
                 ;
1900                   *=DSPAGE
                 ;
                 ;
                 ;
                 ;
                 ;
1900 05     DISLNS .BYTE 5        NUMBER OF LINES TO BE
                 ;                DISASSEMBLED BY TV.DIS.
                 ;
1901 00     LINUM  .BYTE 0        DATA CELL: USED BY TV.DIS.
                 ;
1902 00     LETTER .BYTE 0        COUNTS LETTERS PRINTED IN
                 ;                A MNEMONIC.  USED BY MNEMON.
                 ;
1903 00     TEMP.X .BYTE 0        DATA CELL USED BY MNEMON.
                 ;
1904 0000   SUBPTR .WORD 0        POINTER TO A SUBROUTINE.
                 ;                SET, USED BY MODE.X
                 ;
1906 00     OPBYTS .BYTE 0        DATA CELL: USED BY FINISH.
                 ;
1907 00     OPCHRS .BYTE 0        DATA CELL: USED BY FINISH.
                 ;
1908 10     ADRCOL .BYTE 16       STARTING COLUMN FOR ADDRESS
                 ;                FIELD.  OSI C-IP OWNERS:
                 ;                FOR NARROW FORMAT, SET
                 ;                ADRCOL=$0B.  SEE NOTES
                 ;                IN LISTING FOR ADDRESS MODE
                 ;                SUBROUTINES.)
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; *******************************************
                 ;
                 ;         TV-DISASSEMBLER
                 ;
                 ; *******************************************
                 ;
                 ;
                 ;
                 ;
                 ;
1909 200814 TV.DIS JSR TVT.ON     SELECT SCREEN FOR OUTPUT.
```

```
1750 190C AD0019        LDA DISLNS     INITIALIZE LINE COUNTER WITH
1760 190F 8D0119        STA LINUM      # OF LINES TO DISASSEMBLE.
1770                ;
1780 1912 A9FF          LDA #$FF       SET END ADDRESS TO $FFFF,
1790 1914 8D5415        STA EA         SO NEXTSL WILL ALWAYS
1800 1917 8D5515        STA EA+1       INCREMENT SELECT POINTER.
1810 191A 207214        JSR CR.LF      ADVANCE TO A NEW LINE.
1820                ;
1830 191D 207D19 TVLOOP JSR DSLINE     DISASSEMBLE ONE LINE.
1840 1920 CE0119        DEC LINUM      DONE LAST LINE YET?
1850 1923 D0F8          BNE TVLOOP     IF NOT, DO NEXT ONE.
1860 1925 60            RTS            IF SO, RETURN.
1870                ;
1880                ;
1890                ;
1900                ;
1910                ;
1920                ;
1930                ;
1940                ;
1950                ;
1960                ;
1970                ; ******************************************
1980                ;
1990                ;      PRINTING DISASSEMBLER
2000                ;
2010                ; ******************************************
2020                ;
2030                ;
2040                ;
2050                ;
2060                ;
2070 1926 201A14 PR.DIS JSR PR.OFF     DE-SELECT PRINTER
2080 1929 200814        JSR TVT.ON     SELECT SCREEN FOR OUTPUT.
2090 192C 20E414        JSR PRINT:     DISPLAY TITLE.
2100 192F 7F            .BYTE TEX,CR,LF
2100 1930 0D
2100 1931 0A
2110 1932 20            .BYTE '     PRINTING DISASSEMBLER.'
2110 1933 20
2110 1934 20
2110 1935 20
2110 1936 20
2110 1937 50
2110 1938 52
2110 1939 49
2110 193A 4E
2110 193B 54
2110 193C 49
2110 193D 4E
2110 193E 47
2110 193F 20
2110 1940 44
2110 1941 49
2110 1942 53
2110 1943 41
2110 1944 53
2110 1945 53
```

```
1946 45
1947 4D
1948 42
1949 4C
194A 45
194B 52
194C 2E
                ;
194D 0D                 .BYTE CR,LF,ETX
194E 0A
194F FF
                ;
1950 20E915             JSR SETADS      LET USER SET START, END
                ;                       ADDRESSES OF MEMORY TO BE
                ;                       DISASSEMBLED.
1953 201414             JSR PR.ON       SELECT PRINTER FOR OUTPUT.
1956 20E414             JSR PRINT:
1959 7F                 .BYTE TEX,CR,LF
195A 0D
195B 0A
195C 44                 .BYTE 'DISASSEMBLING '
195D 49
195E 53
195F 41
1960 53
1961 53
1962 45
1963 4D
1964 42
1965 4C
1966 49
1967 4E
1968 47
1969 20
196A FF                 .BYTE ETX
196B 20DF16             JSR RANGE       PRINT RANGE OF MEMORY TO
                ;                       BE DISASSEMBLED.
196E 20A017             JSR GOTOSA      SET SELECT=START OF BLOCK.
                ;
1971 207214             JSR CR.LF       ADVANCE TO A NEW LINE.
1974 207D19 PRLOOP JSR DSLINE           DISASSEMBLE ONE LINE.
1977 10FB               BPL PRLOOP      IF IT WASN'T THE LAST LINE,
                ;                       DISASSEMBLE THE NEXT ONE.
                ;
                ;
1979 201A14             JSR PR.OFF      DE-SELECT PRINTER FOR OUTPUT.
                ;
197C 60                 RTS             RETURN TO CALLER.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
```

```
2460                ; ******************************************
2470                ;
2480                ;         DISASSEMBLE ONE LINE.
2490                ;
2500                ; ******************************************
2510                ;
2520                ;
2530                ;
2540                ;
2550                ;
2560 197D 209412 DSLINE JSR GET.SL      GET CURRENTLY-SELECTED BYTE.
2570 1980 48            PHA             SAVE IT ON STACK.
2580 1981 209219        JSR MNEMON      PRINT MNEMONIC REPRESENTED
2590                ;                   BY THAT OPCODE.
2600 1984 207D14        JSR SPACE       SPACE ONCE.
2610 1987 68            PLA             RESTORE OPCODE.
2620 1988 20AF19        JSR OPERND      PRINT OPERAND REQUIRED BY
2630                ;                   THAT OPCODE.
2640 198B 20011A        JSR FINISH      FINISH THE LINE BY PRINTING
2650                ;                   FIELDS 3-6.  FINISH LEAVES
2660                ;                   SELECT POINTING TO LAST
2670                ;                   BYTE OF INSTRUCTION.
2680                ;
2690 198E 208317        JSR NEXTSL      SELECT NEXT BYTE, IF
2700                ;                   SELECT<EA.
2710 1991 60            RTS             RETURN W/RETURNCODE FROM
2720                ;                   NEXTSL.  SELECT POINTS TO
2730                ;                   NEXT OPCODE, OR SELECT=EA.
2740                ;
2750                ;
2760                ;
2770                ;
2780                ;
2790                ;
2800                ;
2810                ;
2820                ; ******************************************
2830                ;
2840                ;          PRINT MNEMONIC
2850                ;
2860                ; ******************************************
2870                ;
2880                ;
2890                ;
2900                ;
2910                ;
2920 1992 A203   MNEMON LDX #3          WE'LL PRINT THREE LETTERS.
2930 1994 8E0219        STX LETTER
2940 1997 AA            TAX             PREPARE TO USE OPCODE AS AN
2950                ;                   INDEX.
2960                ;
2970 1998 BD001C        LDA MCODES,X    LOOK UP MNEMONIC CODE FOR
2980                ;                   THAT OPCODE.  MCODES IS
2990                ;                   TABLE OF MNEMONIC CODES.
3000                ;
3010 199B AA            TAX             PREPARE TO USE THAT MNEMONIC
3020                ;                   CODE AS AN INDEX.
3030 199C BD501B MNLOOP LDA MNAMES,X    GET A MNEMONIC CHARACTER.
```

```
                 ;                          (MNAMES IS A LIST OF
                 ;                          MNEMONIC NAMES.)
                 ;
19BF 8E0319          STX TEMP.X             SAVE X-REGISTER, SINCE
                 ;                          PRINTING MAY CHANGE X.
19A2 204014          JSR PR.CHR             PRINT THE MNEMONIC CHARACTER.
19A5 AE0319          LDX TEMP.X             RESTORE X,
19A8 E8              INX                    ADJUST INDEX FOR NEXT LETTER.
19A9 CE0219          DEC LETTER             PRINTED 3 LETTERS YET?
19AC D0EE            BNE MNLOOP             IF NOT, PRINT NEXT ONE.
19AE 60              RTS                    IF SO, RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;        PRINT OPERAND
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
19AF AA          OPERND TAX                 LOOK UP ADDRESSING MODE
19B0 BD001D          LDA MODES,X            CODE FOR THIS OPCODE.
                 ;
19B3 AA              TAX                    X NOW INDICATES ADDRESSING
                 ;                          MODE.
                 ;
19B4 20B819          JSR MODE.X             HANDLE THAT ADDRESSING MODE.
19B7 60              RTS                    RETURN TO CALLER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;       HANDLE ADDRESSING MODE "X"
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
```

```
                     ;
                     ;
1988 BD1B1B MODE.X LDA SUBS,X     GET LOW BYTE OF Xth POINTER
19BB 8D0419        STA SUBPTR     IN TABLE OF SUBROUTINE
                     ;            POINTERS.
19BE E8            INX            ADJUST INDEX FOR NEXT BYTE.
19BF BD1B1B        LDA SUBS,X     GET HIGH BYTE OF POINTER.
19C2 8D0519        STA SUBPTR+1
19C5 6C0419        JMP (SUBPTR)   JUMP TO SUBROUTINE SPECIFIED
                     ;            BY SUBROUTINE POINTER.
                     ;            THAT SUBROUTINE WILL RETURN
                     ;            TO THE CALLER OF MODE.X,
                     ;            NOT TO MODE.X ITSELF.
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;
                     ; ******************************************
                     ;
                     ;           DISASSEMBLER UTILITIES
                     ;
                     ; ******************************************
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;           PRINT ONE-BYTE OPERAND
                     ;
                     ;
                     ;
19C9 200D13 ONEBYT JSR INC.SL     ADVANCE TO BYTE FOLLOWING
                     ;            OPCODE.
19CB 209A15        JSR DUMPSL     DUMP THAT BYTE.
19CE 60            RTS            RETURN TO CALLER.
                     ;
                     ;
                     ;
                     ;
                     ;
                     ;           PRINT TWO-BYTE OPERAND:
                     ;
                     ;
                     ;
19CF 200D13 TWOBYT JSR INC.SL     ADVANCE TO FIRST BYTE OF
                     ;            OPERAND.
19D2 209412        JSR GET.SL     LOAD THAT BYTE INTO ACC.
19D5 48            PHA            SAVE IT.
19D6 200D13        JSR INC.SL     ADVANCE TO 2ND BYTE OF
                     ;            OPERAND.
19D9 209A15        JSR DUMPSL     DUMP IT.
19DC 68            PLA            RESTORE FIRST BYTE TO ACC.
```

```
4200 19DD 208314        JSR PR.BYT      DUMP IT.
4210 19E0 60            RTS             RETURN TO CALLER.
4220               ;
4230               ;
4240               ;
4250               ;
4260               ;
4270               ;    PRINT LEFT, RIGHT PARENTHESES
4280               ;
4290               ;
4300               ;
4310 19E1 A928   LPAREN LDA #'(
4320 19E3 D002          BNE SENDIT
4330               ;
4340               ;
4350 19E5 A929   RPAREN LDA #')
4360               ;
4370 19E7 204014 SENDIT JSR PR.CHR
4380 19EA 60            RTS
4390               ;
4400               ;
4410               ;
4420               ;
4430               ;
4440               ;      PRINT A COMMA AND AN "X"
4450               ;
4460               ;
4470               ;
4480 19EB A92C   XINDEX LDA #',
4490 19ED 204014        JSR PR.CHR      PRINT A COMMA.
4500 19F0 A958          LDA #'X
4510 19F2 204014        JSR PR.CHR      PRINT AN "X".
4520 19F5 60            RTS
4530               ;
4540               ;
4550               ;
4560               ;
4570               ;
4580               ;      PRINT A COMMA AND A "Y"
4590               ;
4600               ;
4610               ;
4620 19F6 A92C   YINDEX LDA #',
4630 19F8 204014        JSR PR.CHR      PRINT COMMA.
4640 19FB A959          LDA #'Y
4650 19FD 204014        JSR PR.CHR      PRINT A "Y".
4660 1A00 60            RTS
4670               ;
4680               ;
4690               ;
4700               ;
4710               ;
4720               ;
4730               ;
4740               ;
4750               ;
4760               ;
4770               ; ********************************************
```

```
4780                    ;
4790                    ;       FINISH THE LINE
4800                    ;
4810                    ; ************************************************
4820                    ;
4830                    ;
4840                    ;
4850                    ;          NOTE:      EVERY ADDRESSING MODE
4860                    ;                     SUBROUTINE MUST END BY
4870                    ;                     SETTING X=# OF BYTES IN
4880                    ;                     OPERAND, AND ACC=# OF
4890                    ;                     CHARACTERS IN OPERAND.
4900                    ;
4910                    ;
4920 1A01 8D0719 FINISH STA OPCHRS            SAVE THE LENGTH OF THE
4930 1A04 8E0619        STX OPBYTS            OPERAND, IN CHARACTERS AND
4940                    ;                     IN BYTES.  0 MEANS NO
4950                    ;                     OPERAND.
4960                    ;
4970 1A07 CA            DEX                   IF NECESSARY, DECREMENT THE
4980                    ;                     SELECT POINTER SO IT POINTS
4990 1A08 3006          BMI SEL.OK            TO THE OPCODE.
5000 1A0A 201A13 LOOP.1 JSR DEC.SL
5010 1A0D CA            DEX
5020 1A0E 10FA          BPL LOOP.1
5030                    ;                     NOW SELECT POINTS TO OPCODE.
5040                    ;
5050                    ;
5060 1A10 08     SEL.OK PHP                   SAVE CALLER'S DECIMAL FLAG.
5070 1A11 D8            CLD                   PREPARE FOR BINARY ADDITION.
5080 1A12 38            SEC                   SPACE OVER TO THE COLUMN
5090 1A13 AD0819        LDA ADRCOL            FOR THE ADDRESS FIELD:
5100 1A16 E904          SBC #4                OPERAND FIELD STARTED IN
5110                    ;                     COLUMN 4...
5120 1A18 ED0719        SBC OPCHRS            AND INCLUDES OPCHRS
5130                    ;                     CHARACTERS.
5140 1A1B 28            PLP                   RESTORE CALLER'S DECIMAL FLAG
5150 1A1C AA            TAX
5160 1A1D 209614        JSR SPACES            PRINT ENOUGH SPACES TO
5170                    ;                     REACH ADDRESS COLUMN.
5180 1A20 20A115        JSR PR.ADR            PRINT ADDRESS OF OPCODE.
5190                    ;
5200 1A23 207D14 LOOP.2 JSR SPACE             SPACE ONCE.
5210 1A26 209A15        JSR DUMPSL            DUMP SELECTED BYTE.
5220 1A29 200D13        JSR INC.SL            SELECT NEXT BYTE.
5230 1A2C CE0619        DEC OPBYTS            DUMPED LAST BYTE IN
5240                    ;                     INSTRUCTION?
5250 1A2F 10F2          BPL LOOP.2            IF NOT, DUMP NEXT BYTE.
5260 1A31 201A13        JSR DEC.SL            BACK UP SELECT, SO IT POINTS
5270                    ;                     TO LAST BYTE IN OPERAND.
5280                    ;
5290                    ;
5300                    ;                     IF SO, GO TO A NEW LINE:
5310                    ;
5320 1A34 207214 FINEND JSR CR.LF             HAVING DISASSEMBLED ONE LINE,
5330                    ;                     GO TO A NEW LINE.
5340 1A37 60            RTS                   RETURN TO CALLER.
5350                    ;
```

Appendix C7:

Table-Driven Disassembler (Addressing Mode Subroutines)

```
;       APPENDIX C7: ASSEMBLER LISTING OF
;          TABLE-DRIVEN DISASSEMBLER:
;
;          ADDRESSING MODE SUBROUTINES
;
;
;
;
;
;
;
;    SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM
; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
;
;
;                   BY KEN SKIER
;
;
;
;
;
;
;
;
;
;
;
;
;
;
;
; *******************************************
;
;            CONSTANTS
;
; *******************************************
;
;
;
;
;
000D=         CR = $0D         CARRIAGE RETURN.
;
000A=         LF = $0A         LINE FEED.
;
;
007F=         TEX = $7F        THIS CHARACTER MUST START
;                              ANY MESSAGE.
;
00FF=         ETX = $FF        THIS CHARACTER MUST END
;                              ANY MESSAGE.
;
;
;
;
;
;
```

```
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; *******************************************
                  ;
                  ;        EXTERNAL ADDRESSES
                  ;
                  ; *******************************************
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
     1200=               VMPAGE=$1200   STARTING PAGE OF VISIBLE
                  ;                     MONITOR CODE.
     1205=               SELECT=VMPAGE+5
     1207=               VISMON=VMPAGE+7
     1294=               GET.SL=VMPAGE+$94
     130D=               INC.SL=VMPAGE+$10D
     131A=               DEC.SL=VMPAGE+$11A
                  ;
                  ;
     1400=               PRPAGE=$1400   STARTING PAGE OF PRINT
                  ;                     UTILITIES.
     1440=               PR.CHR=PRPAGE+$40
     1472=               CR.LF =PRPAGE+$72
     147D=               SPACE =PRPAGE+$7D
     1496=               SPACES=PRPAGE+$96
     1483=               PR.BYT=PRPAGE+$83
     14E4=               PRINT:=PRPAGE+$E4
     1512=               PUSHSL=PRPAGE+$112
     152B=               POP.SL=PRPAGE+$12B
                  ;
                  ;
     1500=               HEX.PG=$1500   ADDRESS OF PAGE IN WHICH
                  ;                     HEXDUMP CODE STARTS.
                  ;
     15A1=               PR.ADR=HEX.PG+$A1
     1783=               NEXTSL=HEX.PG+$283
                  ;
                  ;
     1900=               DSPAGE=$1900   START OF DISASSEMBLER CODE.
                  ;
     19C8=               ONEBYT=DSPAGE+$C8
     19CF=               TWOBYT=DSPAGE+$CF
     19E1=               LPAREN=DSPAGE+$E1
     19E5=               RPAREN=DSPAGE+$E5
     19EB=               XINDEX=DSPAGE+$EB
```

```
19F6=              YINDEX=DSPAGE+$F6
               ;
               ;
               ;
               ;
               ;
               ;
               ;
1A40               *=DSPAGE+$140
               ;
               ;
               ;
               ;
               ;
               ;
               ;
               ;
               ;
               ;
               ;
               ; ********************************************
               ;
               ;          ADDRESSING MODE SUBROUTINES
               ;
               ; ********************************************
               ;
               ;
               ;
               ;
               ;
               ;
               ;          ABSOLUTE MODE
               ;
               ;
               ;
1A40 20CF19 ABSLUT JSR TWOBYT      PRINT A TWO-BYTE OPERAND.
1A43 A202          LDX #2          OPERAND HAS TWO BYTES...
1A45 A904          LDA #4          ...AND FOUR CHARACTERS.
1A47 60            RTS             RETURN TO CALLER.
               ;
               ;
               ;
               ;
               ;
               ;          ABSOLUTE,X MODE
               ;
               ;
               ;
1A48 20401A ABS.X  JSR ABSLUT
1A4B 20EB19        JSR XINDEX      PRINT A COMMA AND AN "X".
1A4E A202          LDX #2          OPERAND HAS 2 BYTES...
1A50 A906          LDA #6          ...AND SIX CHARACTERS.
1A52 60            RTS             RETURN TO CALLER.
               ;
               ;
               ;
               ;
               ;
```

```
1740                 ;        ABSOLUTE.Y MODE
1750                 ;
1760                 ;
1770                 ;
1780 1A53 20401A ABS.Y  JSR ABSLUT
1790 1A56 20F619        JSR YINDEX
1800 1A59 A202          LDX #2
1810 1A5B A906          LDA #6
1820 1A5D 60            RTS
1830                 ;
1840                 ;
1850                 ;
1860                 ;
1870                 ;
1880                 ;        ACCUMULATOR MODE
1890                 ;
1900                 ;
1910 1A5E A941   ACC    LDA #'A        PRINT THE LETTER "A"
1920 1A60 204014        JSR PR.CHR
1930 1A63 A200          LDX #0         OPERAND HAS NO BYTES...
1940 1A65 A901          LDA #1         ...AND ONE CHARACTER.
1950 1A67 60            RTS            RETURN TO CALLER.
1960                 ;
1970                 ;
1980                 ;
1990                 ;
2000                 ;
2010                 ;        IMPLIED MODE
2020                 ;
2030                 ;
2040                 ;
2050 1A68 A200   IMPLID LDX #0         OPERAND HAS NO BYTES...
2060 1A6A A900          LDA #0         ...AND NO CHARACTERS.
2070 1A6C 60            RTS
2080                 ;
2090                 ;
2100                 ;
2110                 ;
2120                 ;
2130                 ;        IMMEDIATE MODE
2140                 ;
2150                 ;
2160                 ;
2170 1A6D A923   IMMEDT LDA #'#        PRINT A "#" CHARACTER.
2180 1A6F 204014        JSR PR.CHR
2190                 ;
2200 1A72 A924          LDA #'$        PRINT A DOLLAR SIGN TO
2210 1A74 204014        JSR PR.CHR     INDICATE HEXADECIMAL.
2220 1A77 20C819        JSR ONEBYT     PRINT ONE-BYTE OPERAND IN
2230                 ;                 HEXADECIMAL FORMAT.
2240 1A7A A201          LDX #1         OPERAND HAS ONE BYTE...
2250 1A7C A904          LDA #4         ...AND FOUR CHARACTERS.
2260 1A7E 60            RTS            RETURN TO CALLER.
2270                 ;
2280                 ;
2290                 ;
2300                 ;
2310                 ;
```

```
2320                  ;         INDIRECT MODE
2330                  ;
2340                  ;
2350                  ;
2360 1A7F 20E119 INDRCT JSR LPAREN     PRINT LEFT PARENTHESIS.
2370 1A82 20401A        JSR ABSLUT     PRINT TWO-BYTE OPERAND.
2380 1A85 20E519        JSR RPAREN     PRINT RIGHT PARENTHESIS.
2390 1A88 A906          LDA #6         A HOLDS NUMBER OF CHARACTERS
2400                  ;                IN OPERAND.
2410 1A8A A202          LDX #2         X HOLDS NUMBER OF BYTES IN
2420                  ;                OPERAND.
2430 1A8C 60            RTS            RETURN TO CALLER.
2440                  ;
2450                  ;
2460                  ;
2470                  ;
2480                  ;
2490                  ;         INDIRECT,X MODE
2500                  ;
2510                  ;
2520                  ;
2530 1A8D 20E119 IND.X  JSR LPAREN
2540 1A90 20E81A        JSR ZERO.X     PRINT A ZERO PAGE ADDRESS,
2550                  ;                A COMMA, AND THE LETTER "X".
2560 1A93 20E519        JSR RPAREN
2570 1A96 A201          LDX #1         ONE BYTE IN OPERAND.
2580 1A98 A908          LDA #8         8 CHARACTERS IN OPERAND.
2590                  ;                (C-IP OWNERS: A9 06, NOT
2600                  ;                A9 08, FOR NARROW FORMAT.)
2610 1A9A 60            RTS
2620                  ;
2630                  ;
2640                  ;
2650                  ;
2660                  ;
2670                  ;       INDIRECT,Y MODE
2680                  ;
2690                  ;
2700                  ;
2710 1A9B 20E119 IND.Y  JSR LPAREN
2720 1A9E 20DB1A        JSR ZEROPG     PRINT A ZERO PAGE ADDRESS.
2730 1AA1 20E519        JSR RPAREN
2740 1AA4 20F619        JSR YINDEX     PRINT A COMMA AND A "Y".
2750 1AA7 A201          LDX #1         OPERAND HAS 1 BYTE...
2760 1AA9 A908          LDA #8         ...AND 8 CHARACTERS.
2770                  ;                (C-IP OWNERS: A9 06, NOT
2780                  ;                A9 08, FOR NARROW FORMAT.)
2790 1AAB 60            RTS
2800                  ;
2810                  ;
2820                  ;
2830                  ;
2840                  ;
2850                  ;         RELATIVE MODE
2860                  ;
2870                  ;
2880                  ;
2890 1AAC 200D13 RELATV JSR INC.SL     SELECT NEXT BYTE.
```

```
1AAF 201215          JSR PUSHSL     SAVE SELECT POINTER ON STACK.
1AB2 209412          JSR GET.SL     GET OPERAND BYTE.
1AB5 48              PHA            SAVE IT ON STACK.
1AB6 200D13          JSR INC.SL     INCREMENT SELECT POINTER
              ;                     SO IT POINTS TO NEXT OPCODE.
              ;                     (RELATIVE BRANCHES ARE
              ;                     RELATIVE TO NEXT OPCODE.)
1AB9 68              PLA            RESTORE OPERAND BYTE TO ACC.
1ABA C900            CMP #0         IS IT PLUS OR MINUS?
1ABC 1003            BPL FORWRD     IF PLUS, IT MEANS A FORWARD
              ;                     BRANCH.
              ;
              ;                     OPERAND IS MINUS, SO WE'LL
              ;                     BRANCH BACKWARD.
1ABE CE0612          DEC SELECT+1   BRANCHING BACKWARD IS LIKE
              ;                     BRANCHING FORWARD FROM ONE
              ;                     PAGE LOWER IN MEMORY.
              ;
              ;
1AC1 08       FORWRD PHP            SAVE CALLER'S DECIMAL FLAG.
1AC2 D8              CLD            CLEAR DECIMAL MODE, FOR
              ;                     BINARY ADDITION.
1AC3 18              CLC            PREPARE TO ADD.
1AC4 6D0512          ADC SELECT     ADD OPERAND BYTE TO SELECT.
1AC7 9003            BCC RELEND
1AC9 EE0612          INC SELECT+1
1ACC 8D0512 RELEND STA SELECT       NOW SELECT POINTS TO ADDRESS
              ;                     SPECIFIED BY RELATIVE
              ;                     BRANCH INSTRUCTION.
1ACF 28              PLP            RESTORE CALLER'S DECIMAL
              ;                     FLAG.
1AD0 20A115          JSR PR.ADR     PRINT ADDRESS SPECIFIED
              ;                     BY INSTRUCTION.
1AD3 202B15          JSR POP.SL     RESTORE SELECT=ADDRESS OF
              ;                     OPERAND.
1AD6 A201            LDX #1         OPERAND HAD ONE BYTE...
1AD8 A904            LDA #4         AND FOUR CHARACTERS.
1ADA 60              RTS            RETURN TO CALLER.
              ;
              ;
              ;
              ;
              ;          ZERO PAGE MODE
              ;
              ;
              ;
              ;
1ADB A900     ZEROPG LDA #0         PRINT TWO ASCII ZERO'S TO
1ADD 208314          JSR PR.BYT     ALL SELECTED BYTES.
              ;                     (C-IP OWNERS: SUBSTITUTE NOPS
              ;                     --EA EA EA--FOR JSR PR.BYT,
              ;                     TO GET NARROW FORMAT.
1AE0 20C819          JSR ONEBYT     PRINT ONE-BYTE OPERAND.
1AE3 A201            LDX #1         OPERAND HAS ONE BYTE...
1AE5 A904            LDA #4         ...AND FOUR CHARACTERS.
              ;                     (C-IP OWNERS:A9 02,
              ;                     NOT A9 04, FOR NARROW FORMAT.)
1AE7 60              RTS
```

```
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;        ZERO PAGE, X  MODE
                    ;
                    ;
                    ;
1AE8 20DB1A ZERO.X JSR ZEROPG     PRINT THE ZERO PAGE ADDRESS.
1AEB 20EB19        JSR XINDEX     PRINT A COMMA AND AN "X".
1AEE A201          LDX #1         OPERAND HAS 1 BYTE...
1AF0 A906          LDA #6         ...AND SIX CHARACTERS.
                    ;             (C-IP OWNERS: A9 04,
                    ;             NOT A9 06, FOR NARROW FORMAT.)
1AF2 60            RTS            RETURN TO CALLER.
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;        ZERO PAGE ,Y MODE
                    ;
                    ;
                    ;
1AF3 20DB1A ZERO.Y JSR ZEROPG
1AF6 20F619        JSR YINDEX
1AF9 A201          LDX #1
1AFB A906          LDA #6         (C-IP OWNERS: A9 04 HERE
                    ;             FOR NARROW FORMAT.)
1AFD 60            RTS
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ; ********************************************
                    ;
                    ;       A PSEUDO-ADDRESSING MODE
                    ;     FOR EMBEDDED TEXT: TEXT MODE.
                    ;
                    ; ********************************************
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;     THE PSEUDO-OPCODE TEX ($7F) BEGINS ANY
                    ; STRING OF TEXT AND PRINT CONTROL CHARACTERS.
                    ; THE PSEUDO-TEXT CHARACTER ETX ($FF) ENDS ANY
```

```
                  ; SUCH STRING.  TEX HAS A PSEUDO-ADDRESSING
                  ; MODE: TEXT MODE.  IN TEXT MODE, WE PRINT THE
                  ; STRING AND RETURN, WITHOUT DUMPING THE LINE
                  ; IN HEX.  THE STRING MAY BE OF ANY LENGTH.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
1AFE 68          TXMODE PLA             POP RETURN ADDRESS TO
1AFF 68                 PLA             OPERND.
                  ;
1B00 68                 PLA             POP RETURN ADDRESS TO
1B01 68                 PLA             DSLINE.
                  ;
                  ;                     NOW DSLINE'S CALLER IS ON
                  ;                     THE STACK.
                  ;
                  ;
1B02 208317             JSR NEXTSL      ADVANCE PAST TEX PSEUDO-OP.
1B05 300D               BMI TXEXIT      RETURN IF REACHED EA.
1B07 209412             JSR GET.SL      GET THE CHARACTER.
1B0A C9FF               CMP #ETX        IS IT END OF TEXT?
1B0C F006               BEQ TXEXIT      IF SO, STRING ENDED.
1B0E 204014             JSR PR.CHR      IF NOT, PRINT CHARACTER.
1B11 18                 CLC             BRANCH BACK TO GET NEXT
1B12 90EE               BCC TXMODE+4    CHARACTER.
                  ;
                  ;
1B14 207214      TXEXIT JSR CR.LF       ADVANCE TO A NEW LINE.
1B17 208317             JSR NEXTSL      ADVANCE TO NEXT OPCODE.
1B1A 60                 RTS             RETURN TO CALLER OF DSLINE.
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;     TABLE OF ADDRESSING MODE SUBROUTINES
                  ;
                  ; ********************************************
                  ;
                  ;
                  ;
                  ;
                  ;
```

```
4640 1B1B 681A   SUBS   .WORD IMPLID   ADDRESSING MODE 0 IS INVALID,
4650                ;                  HENCE IMPLIED.
4660 1B1D 5E1A          .WORD ACC
4670 1B1F 6D1A          .WORD IMMEDT
4680 1B21 DB1A          .WORD ZEROPG
4690 1B23 E81A          .WORD ZERO.X
4700 1B25 F31A          .WORD ZERO.Y
4710 1B27 401A          .WORD ABSLUT
4720 1B29 481A          .WORD ABS.X
4730 1B2B 531A          .WORD ABS.Y
4740 1B2D 681A          .WORD IMPLID
4750 1B2F AC1A          .WORD RELATV
4760 1B31 8D1A          .WORD IND.X
4770 1B33 9B1A          .WORD IND.Y
4780 1B35 7F1A          .WORD INDRCT
4790 1B37 FE1A          .WORD TXMODE
```

Appendix C8:

Table-Driven Disassembler (Tables)

```
          ;      APPENDIX C8: ASSEMBLER LISTING OF
          ;         TABLE-DRIVEN DISASSEMBLER
          ;
          ;                    TABLES
          ;
          ;
          ;
          ;
          ;     SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM
          ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
          ;
          ;
          ;                  BY KEN SKIER
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ; ********************************************
          ;
          ;           CONSTANTS
          ;
          ; ********************************************
          ;
          ;
          ;
          ;
          ;
          ;
          ;
007F=           TEX = $7F       THIS CHARACTER MUST START
          ;                     ANY MESSAGE.
          ;
00FF=           ETX = $FF       THIS CHARACTER MUST END
          ;                     ANY MESSAGE.
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
```

```
580                  ;
590                  ;
600 1900=                 DSPAGE=$1900   STARTING PAGE OF DISASSEMBLER
610                  ;
620                  ;
630                  ;
640                  ;
650                  ;
660                  ; *********************************************
670                  ;
680                  ;           LIST OF MNEMONICS
690                  ;
700                  ; *********************************************
710                  ;
720                  ;
730                  ;
740 1B50                  *=DSPAGE+$250
750                  ;
760                  ;
770                  ;
780                  ;
790                  ;
800 1B50 7F          MNAMES .BYTE TEX        SINCE THIS TABLE IS A
810                  ;                       STRING OF CHARACTERS, START
820                  ;                       IT WITH THE TEX PSEUDO-OP.
830                  ;
840 1B51 42                 .BYTE 'BAD'
840 1B52 41
840 1B53 44
850 1B54 41                 .BYTE 'ADC'
850 1B55 44
850 1B56 43
860 1B57 41                 .BYTE 'AND'
860 1B58 4E
860 1B59 44
870 1B5A 41                 .BYTE 'ASL'
870 1B5B 53
870 1B5C 4C
880 1B5D 42                 .BYTE 'BCC'
880 1B5E 43
880 1B5F 43
890 1B60 42                 .BYTE 'BCS'
890 1B61 43
890 1B62 53
900 1B63 42                 .BYTE 'BEQ'
900 1B64 45
900 1B65 51
910 1B66 42                 .BYTE 'BIT'
910 1B67 49
910 1B68 54
920 1B69 42                 .BYTE 'BMI'
920 1B6A 4D
920 1B6B 49
930 1B6C 42                 .BYTE 'BNE'
930 1B6D 4E
930 1B6E 45
940 1B6F 42                 .BYTE 'BPL'
940 1B70 50
```

```
 940 1B71 4C
 950 1B72 42            .BYTE 'BRK'
 950 1B73 52
 950 1B74 4B
 960 1B75 42            .BYTE 'BVC'
 960 1B76 56
 960 1B77 43
 970 1B78 42            .BYTE 'BVS'
 970 1B79 56
 970 1B7A 53
 980 1B7B 43            .BYTE 'CLC'
 980 1B7C 4C
 980 1B7D 43
 990 1B7E 43            .BYTE 'CLD'
 990 1B7F 4C
 990 1B80 44
1000 1B81 43            .BYTE 'CLI'
1000 1B82 4C
1000 1B83 49
1010 1B84 43            .BYTE 'CLV'
1010 1B85 4C
1010 1B86 56
1020 1B87 43            .BYTE 'CMP'
1020 1B88 4D
1020 1B89 50
1030 1B8A 43            .BYTE 'CPX'
1030 1B8B 50
1030 1B8C 58
1040 1B8D 43            .BYTE 'CPY'
1040 1B8E 50
1040 1B8F 59
1050 1B90 44            .BYTE 'DEC'
1050 1B91 45
1050 1B92 43
1060 1B93 44            .BYTE 'DEX'
1060 1B94 45
1060 1B95 58
1070 1B96 44            .BYTE 'DEY'
1070 1B97 45
1070 1B98 59
1080 1B99 45            .BYTE 'EOR'
1080 1B9A 4F
1080 1B9B 52
1090 1B9C 49            .BYTE 'INC'
1090 1B9D 4E
1090 1B9E 43
1100 1B9F 49            .BYTE 'INX'
1100 1BA0 4E
1100 1BA1 58
1110 1BA2 49            .BYTE 'INY'
1110 1BA3 4E
1110 1BA4 59
1120 1BA5 4A            .BYTE 'JMP'
1120 1BA6 4D
1120 1BA7 50
1130 1BA8 4A            .BYTE 'JSR'
1130 1BA9 53
1130 1BAA 52
```

```
1140 1BAB 4C          .BYTE 'LDA'
1140 1BAC 44
1140 1BAD 41
1150 1BAE 4C          .BYTE 'LDX'
1150 1BAF 44
1150 1BB0 58
1160 1BB1 4C          .BYTE 'LDY'
1160 1BB2 44
1160 1BB3 59
1170 1BB4 4C          .BYTE 'LSR'
1170 1BB5 53
1170 1BB6 52
1180 1BB7 4E          .BYTE 'NOP'
1180 1BB8 4F
1180 1BB9 50
1190 1BBA 4F          .BYTE 'ORA'
1190 1BBB 52
1190 1BBC 41
1200 1BBD 50          .BYTE 'PHA'
1200 1BBE 48
1200 1BBF 41
1210 1BC0 50          .BYTE 'PHP'
1210 1BC1 48
1210 1BC2 50
1220 1BC3 50          .BYTE 'PLA'
1220 1BC4 4C
1220 1BC5 41
1230 1BC6 50          .BYTE 'PLP'
1230 1BC7 4C
1230 1BC8 50
1240 1BC9 52          .BYTE 'ROL'
1240 1BCA 4F
1240 1BCB 4C
1250 1BCC 52          .BYTE 'ROR'
1250 1BCD 4F
1250 1BCE 52
1260 1BCF 52          .BYTE 'RTI'
1260 1BD0 54
1260 1BD1 49
1270 1BD2 52          .BYTE 'RTS'
1270 1BD3 54
1270 1BD4 53
1280 1BD5 53          .BYTE 'SBC'
1280 1BD6 42
1280 1BD7 43
1290 1BD8 53          .BYTE 'SEC'
1290 1BD9 45
1290 1BDA 43
1300 1BDB 53          .BYTE 'SED'
1300 1BDC 45
1300 1BDD 44
1310 1BDE 53          .BYTE 'SEI'
1310 1BDF 45
1310 1BE0 49
1320 1BE1 53          .BYTE 'STA'
1320 1BE2 54
1320 1BE3 41
1330 1BE4 53          .BYTE 'STX'
```

```
1330 1BE5 54
1330 1BE6 58
1340 1BE7 53          .BYTE 'STY'
1340 1BE8 54
1340 1BE9 59
1350 1BEA 54          .BYTE 'TAX'
1350 1BEB 41
1350 1BEC 58
1360 1BED 54          .BYTE 'TAY'
1360 1BEE 41
1360 1BEF 59
1370 1BF0 54          .BYTE 'TSX'
1370 1BF1 53
1370 1BF2 58
1380 1BF3 54          .BYTE 'TXA'
1380 1BF4 58
1380 1BF5 41
1390 1BF6 54          .BYTE 'TXS'
1390 1BF7 58
1390 1BF8 53
1400 1BF9 54          .BYTE 'TYA'
1400 1BFA 59
1400 1BFB 41
1410 1BFC 54          .BYTE 'TEX'
1410 1BFD 45
1410 1BFE 58
1420             ;
1430 1BFF FF          .BYTE ETX      SINCE THIS IS THE END OF A
1440             ;                   STRING OF CHARACTERS, USE
1450             ;                   ETX TO INDICATE END OF TEXT.
1460             ;
1470             ;
1480             ;
1490             ;
1500             ;
1510             ;
1520             ;
1530             ;
1540             ;
1550             ;
1560             ;
1570             ; ********************************************
1580             ;
1590             ;         TABLE OF MNEMONIC CODES
1600             ;
1610             ; ********************************************
1620             ;
1630             ;
1640             ;
1650             ;
1660             ;
1670             ;    A MNEMONIC'S CODE IS ITS OFFSET INTO
1680             ;  MNAMES, THE LIST OF MNEONIC NAMES.
1690             ;
1700             ;
1710             ;
1720 1C00 22     MCODES .BYTE $22,$6A,1,1,1,$6A,$0A,1,$70
1720 1C01 6A
```

```
1720 1C02 01
1720 1C03 01
1720 1C04 01
1720 1C05 6A
1720 1C06 0A
1720 1C07 01
1720 1C08 70
1730 1C09 6A                 .BYTE $6A,$0A,1,1,$6A,$0A,1
1730 1C0A 0A
1730 1C0B 01
1730 1C0C 01
1730 1C0D 6A
1730 1C0E 0A
1730 1C0F 01
1740 1C10 1F                 .BYTE $1F,$6A,1,1,1,$6A,$0A,1
1740 1C11 6A
1740 1C12 01
1740 1C13 01
1740 1C14 01
1740 1C15 6A
1740 1C16 0A
1740 1C17 01
1750 1C18 2B                 .BYTE $2B,$6A,1,1,1,$6A,$0A,1
1750 1C19 6A
1750 1C1A 01
1750 1C1B 01
1750 1C1C 01
1750 1C1D 6A
1750 1C1E 0A
1750 1C1F 01
1760 1C20 58                 .BYTE $58,7,1,1,$16,7,$79,1
1760 1C21 07
1760 1C22 01
1760 1C23 01
1760 1C24 16
1760 1C25 07
1760 1C26 79
1760 1C27 01
1770 1C28 76                 .BYTE $76,7,$79,1,$16,7,$79,1
1770 1C29 07
1770 1C2A 79
1770 1C2B 01
1770 1C2C 16
1770 1C2D 07
1770 1C2E 79
1770 1C2F 01
1780 1C30 19                 .BYTE $19,7,1,1,1,7,$79,1
1780 1C31 07
1780 1C32 01
1780 1C33 01
1780 1C34 01
1780 1C35 07
1780 1C36 79
1780 1C37 01
1790 1C38 88                 .BYTE $88,7,1,1,1,7,$79,1
1790 1C39 07
1790 1C3A 01
1790 1C3B 01
```

```
1790 1C3C 01
1790 1C3D 07
1790 1C3E 79
1790 1C3F 01
1800 1C40 7F          .BYTE $7F,$49,1,1,1,$49,$64,1
1800 1C41 49
1800 1C42 01
1800 1C43 01
1800 1C44 01
1800 1C45 49
1800 1C46 64
1800 1C47 01
1810 1C48 6D          .BYTE $6D,$49,$64,1,$55,$49,$64,1
1810 1C49 49
1810 1C4A 64
1810 1C4B 01
1810 1C4C 55
1810 1C4D 49
1810 1C4E 64
1810 1C4F 01
1820 1C50 25          .BYTE $25,$49,1,1,1,$49,$64,1
1820 1C51 49
1820 1C52 01
1820 1C53 01
1820 1C54 01
1820 1C55 49
1820 1C56 64
1820 1C57 01
1830 1C58 31          .BYTE $31,$49,1,1,1,$49,$64,1
1830 1C59 49
1830 1C5A 01
1830 1C5B 01
1830 1C5C 01
1830 1C5D 49
1830 1C5E 64
1830 1C5F 01
1840 1C60 82          .BYTE $82,4,1,1,1,4,$7C,1
1840 1C61 04
1840 1C62 01
1840 1C63 01
1840 1C64 01
1840 1C65 04
1840 1C66 7C
1840 1C67 01
1850 1C68 73          .BYTE $73,4,$7C,1,$55,4,$7C,1
1850 1C69 04
1850 1C6A 7C
1850 1C6B 01
1850 1C6C 55
1850 1C6D 04
1850 1C6E 7C
1850 1C6F 01
1860 1C70 28          .BYTE $28,4,1,1,1,4,$7C,1
1860 1C71 04
1860 1C72 01
1860 1C73 01
1860 1C74 01
1860 1C75 04
```

```
1860 1C76 7C
1860 1C77 01
1870 1C78 8E                 .BYTE $8E,4,1,1,1,4,$7C,$AC
1870 1C79 04
1870 1C7A 01
1870 1C7B 01
1870 1C7C 01
1870 1C7D 04
1870 1C7E 7C
1870 1C7F AC
1880 1C80 01                 .BYTE 1,$91,1,1,$97,$91,$94,1
1880 1C81 91
1880 1C82 01
1880 1C83 01
1880 1C84 97
1880 1C85 91
1880 1C86 94
1880 1C87 01
1890 1C88 46                 .BYTE $46,1,$A3,1,$97,$91,$94,1
1890 1C89 01
1890 1C8A A3
1890 1C8B 01
1890 1C8C 97
1890 1C8D 91
1890 1C8E 94
1890 1C8F 01
1900 1C90 0D                 .BYTE $0D,$91,1,1,$97,$91,$94,1
1900 1C91 91
1900 1C92 01
1900 1C93 01
1900 1C94 97
1900 1C95 91
1900 1C96 94
1900 1C97 01
1910 1C98 A9                 .BYTE $A9,$91,$A3,1,1,$91,1,1
1910 1C99 91
1910 1C9A A3
1910 1C9B 01
1910 1C9C 01
1910 1C9D 91
1910 1C9E 01
1910 1C9F 01
1920 1CA0 61                 .BYTE $61,$5B,$5E,1,$61,$5B,$5E,1
1920 1CA1 5B
1920 1CA2 5E
1920 1CA3 01
1920 1CA4 61
1920 1CA5 5B
1920 1CA6 5E
1920 1CA7 01
1930 1CA8 9D                 .BYTE $9D,$5B,$9A,1,$61,$5B,$5E,1
1930 1CA9 5B
1930 1CAA 9A
1930 1CAB 01
1930 1CAC 61
1930 1CAD 5B
1930 1CAE 5E
1930 1CAF 01
```

```
1940 1CB0 10            .BYTE $10,$5B,1,1,$61,$5B,$5E,1
1940 1CB1 5B
1940 1CB2 01
1940 1CB3 01
1940 1CB4 61
1940 1CB5 5B
1940 1CB6 5E
1940 1CB7 01
1950 1CB8 34            .BYTE $34,$5B,$9E,1,$61,$5B,$5E,1
1950 1CB9 5B
1950 1CBA 9E
1950 1CBB 01
1950 1CBC 61
1950 1CBD 5B
1950 1CBE 5E
1950 1CBF 01
1560 1CC0 3D            .BYTE $3D,$37,1,1,$3D,$37,$40,1
1960 1CC1 37
1960 1CC2 01
1960 1CC3 01
1960 1CC4 3D
1960 1CC5 37
1960 1CC6 40
1960 1CC7 01
1970 1CC8 52            .BYTE $52,$37,$43,1,$3D,$37,$40,1
1970 1CC9 37
1970 1CCA 43
1970 1CCB 01
1970 1CCC 3D
1970 1CCD 37
1970 1CCE 40
1970 1CCF 01
1980 1CD0 1C            .BYTE $1C,$37,1,1,1,$37,$40,1
1980 1CD1 37
1980 1CD2 01
1980 1CD3 01
1980 1CD4 01
1980 1CD5 37
1980 1CD6 40
1980 1CD7 01
1990 1CD8 2E            .BYTE $2E,$37,1,1,1,$37.$40,1
1990 1CD9 37
1990 1CDA 01
1990 1CDB 01
1990 1CDC 01
1990 1CDD 37
1990 1CDE 40
1990 1CDF 01
2000 1CE0 3A            .BYTE $3A,$85.1,1,$3A,$85,$4C,1
2000 1CE1 85
2000 1CE2 01
2000 1CE3 01
2000 1CE4 3A
2000 1CE5 85
2000 1CE6 4C
2000 1CE7 01
2010 1CE8 4F            .BYTE $4F,$85.$67,1,$3A,$85,$4C.1
2010 1CE9 85
```

```
2010 1CEA 67
2010 1CEB 01
2010 1CEC 3A
2010 1CED 85
2010 1CEE 4C
2010 1CEF 01
2020 1CF0 13          .BYTE $13,$85,1,1,1,$85,$4C,1
2020 1CF1 85
2020 1CF2 01
2020 1CF3 01
2020 1CF4 01
2020 1CF5 85
2020 1CF6 4C
2020 1CF7 01
2030 1CF8 8B          .BYTE $8B,$85,1,1,1,$85,$4C,1
2030 1CF9 85
2030 1CFA 01
2030 1CFB 01
2030 1CFC 01
2030 1CFD 85
2030 1CFE 4C
2030 1CFF 01
2040               ;
2050               ;
2060               ;
2070               ;
2080               ;
2090               ;
2100               ;
2110               ;
2120               ;
2130               ;
2140               ;
2150               ; ********************************************
2160               ;
2170               ;        TABLE OF ADDRESSING MODE CODES
2180               ;
2190               ; ********************************************
2200               ;
2210               ;
2220               ;
2230               ;
2240               ;     AN ADDRESSING MODE'S CODE IS ITS OFFSET
2250               ; INTO SUBS, THE TABLE OF ADDRESSING MODE
2260               ; SUBROUTINES.
2270               ;
2280               ;
2290               ;
2300               ;
2310               ;
2320               ;
2330 1D00 12       MODES  .BYTE 18,22,0,0,0,6,6,0
2330 1D01 16
2330 1D02 00
2330 1D03 00
2330 1D04 00
2330 1D05 06
2330 1D06 06
```

```
2330 1D07 00
2340 1D08 12              .BYTE 18,4,2,0,0,12,12,0
2340 1D09 04
2340 1D0A 02
2340 1D0B 00
2340 1D0C 00
2340 1D0D 0C
2340 1D0E 0C
2340 1D0F 00
2350 1D10 14              .BYTE 20,24,0,0,0,14,14,0
2350 1D11 18
2350 1D12 00
2350 1D13 00
2350 1D14 00
2350 1D15 0E
2350 1D16 0E
2350 1D17 00
2360 1D18 12              .BYTE 18,16,0,0,0,22,22,0
2360 1D19 10
2360 1D1A 00
2360 1D1B 00
2360 1D1C 00
2360 1D1D 16
2360 1D1E 16
2360 1D1F 00
2370 1D20 0C              .BYTE 12,22,0,0,6,6,6,0
2370 1D21 16
2370 1D22 00
2370 1D23 00
2370 1D24 06
2370 1D25 06
2370 1D26 06
2370 1D27 00
2380 1D28 12              .BYTE 18,4,2,0,12,12,12,0
2380 1D29 04
2380 1D2A 02
2380 1D2B 00
2380 1D2C 0C
2380 1D2D 0C
2380 1D2E 0C
2380 1D2F 00
2390 1D30 14              .BYTE 20,24,0,0,0,8,8,0
2390 1D31 18
2390 1D32 00
2390 1D33 00
2390 1D34 00
2390 1D35 08
2390 1D36 08
2390 1D37 00
2400 1D38 12              .BYTE 18,16,0,0,0,14,14,0
2400 1D39 10
2400 1D3A 00
2400 1D3B 00
2400 1D3C 00
2400 1D3D 0E
2400 1D3E 0E
2400 1D3F 00
2410 1D40 12              .BYTE 18,22,0,0,0,6,6,0
```

```
2410 1D41 16
2410 1D42 00
2410 1D43 00
2410 1D44 00
2410 1D45 06
2410 1D46 06
2410 1D47 00
2420 1D48 12          .BYTE 18,12,2,0,12,12,12,0
2420 1D49 0C
2420 1D4A 02
2420 1D4B 00
2420 1D4C 0C
2420 1D4D 0C
2420 1D4E 0C
2420 1D4F 00
2430 1D50 14          .BYTE 20,24,0,0,0,8,8,0
2430 1D51 18
2430 1D52 00
2430 1D53 00
2430 1D54 00
2430 1D55 08
2430 1D56 08
2430 1D57 00
2440 1D58 12          .BYTE 18,16,0,0,0,14,14,0
2440 1D59 10
2440 1D5A 00
2440 1D5B 00
2440 1D5C 00
2440 1D5D 0E
2440 1D5E 0E
2440 1D5F 00
2450 1D60 12          .BYTE 18,22,0,0,0,6,6,0
2450 1D61 16
2450 1D62 00
2450 1D63 00
2450 1D64 00
2450 1D65 06
2450 1D66 06
2450 1D67 00
2460 1D68 12          .BYTE 18,4,2,0,26,12,12,0
2460 1D69 04
2460 1D6A 02
2460 1D6B 00
2460 1D6C 1A
2460 1D6D 0C
2460 1D6E 0C
2460 1D6F 00
2470 1D70 14          .BYTE 20.24,0,0.0,8.8,0
2470 1D71 18
2470 1D72 00
2470 1D73 00
2470 1D74 00
2470 1D75 08
2470 1D76 08
2470 1D77 00
2480 1D78 12          .BYTE 18,16,0,0.0,14,14,28
2480 1D79 10
2480 1D7A 00
```

```
2480 1D7B 00
2480 1D7C 00
2480 1D7D 0E
2480 1D7E 0E
2480 1D7F 1C
2490
2500 1D80 00            .BYTE 0,22,0,0,6,6,6,0
2500 1D81 16
2500 1D82 00
2500 1D83 00
2500 1D84 06
2500 1D85 06
2500 1D86 06
2500 1D87 00
2510 1D88 12            .BYTE 18,0,18,0,12,12,12,0
2510 1D89 00
2510 1D8A 12
2510 1D8B 00
2510 1D8C 0C
2510 1D8D 0C
2510 1D8E 0C
2510 1D8F 00
2520 1D90 14            .BYTE 20,24,0,0,8,8,10,0
2520 1D91 18
2520 1D92 00
2520 1D93 00
2520 1D94 08
2520 1D95 08
2520 1D96 0A
2520 1D97 00
2530 1D98 12            .BYTE 18,16,18,0,0,14,0.0
2530 1D99 10
2530 1D9A 12
2530 1D9B 00
2530 1D9C 00
2530 1D9D 0E
2530 1D9E 00
2530 1D9F 00
2540 1DA0 04            .BYTE 4,22,4,0,6,6,6,0
2540 1DA1 16
2540 1DA2 04
2540 1DA3 00
2540 1DA4 06
2540 1DA5 06
2540 1DA6 06
2540 1DA7 00
2550 1DA8 12            .BYTE 18,4,18,0,12,12,12.0
2550 1DA9 04
2550 1DAA 12
2550 1DAB 00
2550 1DAC 0C
2550 1DAD 0C
2550 1DAE 0C
2550 1DAF 00
2560 1DB0 14            .BYTE 20,24,0.0,8,8,10,0
2560 1DB1 18
2560 1DB2 00
2560 1DB3 00
```

```
2560 1DB4 08
2560 1DB5 08
2560 1DB6 0A
2560 1DB7 00
2570 1DB8 14          .BYTE 20,16,18,0,14,14,16,0
2570 1DB9 10
2570 1DBA 12
2570 1DBB 00
2570 1DBC 0E
2570 1DBD 0E
2570 1DBE 10
2570 1DBF 00
2580 1DC0 04          .BYTE 4,22,0,0,6,6,6,0
2580 1DC1 16
2580 1DC2 00
2580 1DC3 00
2580 1DC4 06
2580 1DC5 06
2580 1DC6 06
2580 1DC7 00
2590 1DC8 12          .BYTE 18,4,18,0,12,12,12,0
2590 1DC9 04
2590 1DCA 12
2590 1DCB 00
2590 1DCC 0C
2590 1DCD 0C
2590 1DCE 0C
2590 1DCF 00
2600 1DD0 14          .BYTE 20,24,0,0,0,8,8,0
2600 1DD1 18
2600 1DD2 00
2600 1DD3 00
2600 1DD4 00
2600 1DD5 08
2600 1DD6 08
2600 1DD7 00
2610 1DD8 12          .BYTE 18,16,0,0,0,14,14,0
2610 1DD9 10
2610 1DDA 00
2610 1DDB 00
2610 1DDC 00
2610 1DDD 0E
2610 1DDE 0E
2610 1DDF 00
2620 1DE0 04          .BYTE 4,22,0,0,6,6,6,0
2620 1DE1 16
2620 1DE2 00
2620 1DE3 00
2620 1DE4 06
2620 1DE5 06
2620 1DE6 06
2620 1DE7 00
2630 1DE8 12          .BYTE 18,4,18,0,12,12,12,0
2630 1DE9 04
2630 1DEA 12
2630 1DEB 00
2630 1DEC 0C
2630 1DED 0C
```

```
2630 1DEE 0C
2630 1DEF 00
2640 1DF0 14          .BYTE 20,24,0,0,0,8,8,0
2640 1DF1 18
2640 1DF2 00
2640 1DF3 00
2640 1DF4 00
2640 1DF5 08
2640 1DF6 08
2640 1DF7 00
2650 1DF8 12          .BYTE 18,16.0,0,0.14.14,0
2650 1DF9 10
2650 1DFA 00
2650 1DFB 00
2650 1DFC 00
2650 1DFD 0E
2650 1DFE 0E
2650 1DFF 00
```

Appendix C9:

Move Utilities

```
                 ;      APPENDIX C9: ASSEMBLER LISTING OF
                 ;                   MOVE UTILITIES
                 ;
                 ;
                 ;
                 ;      SEE CHAPTER 10 OF BEYOND GAMES: SYSTEMS
                 ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.
                 ;
                 ;                   BY KEN SKIER
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;         CONSTANTS
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
    000D=               CR=$0D         CARRIAGE RETURN.
    000A=               LF=$0A         LINE FEED.
    007F=               TEX=$7F        START OF TEXT CHARACTER.
    00FF=               ETX=$FF        END OF TEXT CHARACTER.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;         EXTERNAL ADDRESSES
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
    1200=               VMPAGE=$1200   STARTING PAGE OF VISIBLE
                 ;                     MONITOR CODE.
                 ;
    1205=               SELECT=VMPAGE+5
    1207=               VISMON=VMPAGE+7
                 ;
                 ;
                 ;
    1400=               PRPAGE=$1400   STARTING PAGE OF PRINT CODE.
                 ;
```

```
1408=                   TVT.ON=PRPAGE+8
14E4=                   PRINT:=PRPAGE+$E4
1512=                   PUSHSL=PRPAGE+$112
152B=                   POP.SL=PRPAGE+$12B
                ;
                ;
1500=                   HEX.PG=$1500    ADDRESS OF PAGE IN WHICH
                ;                       HEXDUMP CODE STARTS.
                ;                       (HEXDUMP CODE STARTS AT
                ;                       $1550, BUT IT'S EASIER TO
                ;                       COUNT FROM $1500.)
                ;
15E9=                   SETADS=HEX.PG+$E9
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;     VARIABLES
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;
17B0                    *=$17B0
                ;
                ;
1552=                   SA=HEX.PG+$52   POINTER TO START ADDRESS
                ;                       OF BLOCK TO BE MOVED.
                ;
1554=                   EA=SA+2         POINTER TO END OF BLOCK TO
                ;                       BE MOVED.
                ;
17B0 0000       NUM     .WORD 0         NUMBER OF BYTES IN BLOCK
                ;                       TO BE MOVED.  ZERO MEANS
                ;                       BLOCK CONTAINS 1 BYTE.
                ;
                ;
17B2 0000       DEST    .WORD 0         POINTER TO BLOCK'S
                ;                       DESTINATION.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
0000=                   GETPTR=0        THESE TWO "PAGE POINTERS"
0002=                   PUTPTR=GETPTR+2 GET AND PUT BYTES.
                ;
                ;
```

```
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;          MOVE TOOL
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;
                ;
                ;
17B4 200814 MOVER  JSR TVT.ON      SELECT SCREEN FOR OUTPUT.
17B7 20E414        JSR PRINT:      DISPLAY A TITLE.
17BA 7F            .BYTE TEX,CR,LF
17BB 0D
17BC 0A
17BD 20            .BYTE '     MOVE TOOL.'
17BE 20
17BF 20
17C0 20
17C1 20
17C2 4D
17C3 4F
17C4 56
17C5 45
17C6 20
17C7 54
17C8 4F
17C9 4F
17CA 4C
17CB 2E
17CC 0D            .BYTE CR,LF,LF,ETX
17CD 0A
17CE 0A
17CF FF
                ;
17D0 20E915        JSR SETADS      GET START ADDRESS, END
                ;                  ADDRESS FROM USER.
                ;
17D3 20B918        JSR SET.DA      GET DESTINATION ADDRESS
                ;                  FROM USER.
                ;                  WITH THOSE POINTERS SET,
                ;                  WE'RE READY TO EXECUTE MOV.EA:
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
```

```
                    ;
                    ;  MOV.EA:     MOVE BLOCK SPECIFIED BY SA, EA, DEST
                    ;
                    ; *********************************************
                    ;
                    ;
                    ;
                    ;
                    ;
                    ;  RETURN CODES:
                    ;
                    ;
0000=                      ERROR=0          THIS RETURN CODE MEANS
                    ;                       SA < EA, SO MOVE ABORTED.
00FF=                      OKAY=$FF         THIS RETURN CODE MEANS
                    ;                       MOVE ACCOMPLISHED.
                    ;
                    ;
17D6 AE5515 MOV.EA LDX EA+1         SET NUM = EA - SA:
17D9 38            SEC
17DA AD5415        LDA EA
17DD ED5215        SBC SA
17E0 8DB017        STA NUM
17E3 B002          BCS MOVE.1
17E5 CA            DEX
17E6 38            SEC
17E7 8A     MOVE.1 TXA
17E8 ED5315        SBC SA+1
17EB 8DB117        STA NUM+1
17EE B003          BCS MOVNUM
                    ;
17F0 A900   ER.RTN LDA #ERROR       IF EA < SA,
17F2 60            RTS              RETURN WITH ERROR CODE.
                    ;
                    ;
                    ;
                    ;
                    ; *********************************************
                    ;
                    ; MOVNUM: MOVE BLOCK SPECIFIED BY SA, NUM, DEST.
                    ;
                    ; *********************************************
                    ;
                    ;
                    ;
17F3 A003   MOVNUM LDY #3           SAVE ZERO PAGE BYTES THAT
17F5 B90000 LOOP.1 LDA GETPTR,Y     WILL BE CHANGED.
17F8 48            PHA
17F9 88            DEY
17FA 10F9          BPL LOOP.1
                    ;
                    ;
17FC 38            SEC              IF DEST>SA, BRANCH TO MOVE-UP
17FD AD5315        LDA SA+1
1800 CDB317        CMP DEST+1
1803 9040          BCC MOVEUP
1805 D018          BNE MOVEDN
                    ;               IF DEST<SA, BRANCH T
```

```
2180                 ;                       MOVE-DOWN.
2190 1807 AD5215            LDA SA
2200 180A CDB217            CMP DEST
2210 180D 9036              BCC MOVEUP
2220 180F D00E              BNE MOVEDN       IF DEST=SA,
2230 1811 A000      OK.RTN  LDY #0           RETURN BEARING "OKAY" CODE.
2240                 ;                       RESTORE ZERO PAGE BYTES
2250 1813 68        LOOP.2  PLA              THAT WERE CHANGED.
2260 1814 990000            STA GETPTR,Y
2270 1817 C8                INY
2280 1818 C004              CPY #4
2290 181A D0F7              BNE LOOP.2
2300 181C A9FF              LDA #OKAY        RETURN W/"OKAY" CODE.
2310 181E 60                RTS
2320                 ;
2330                 ;
2340                 ;
2350 181F 20A418    MOVEDN  JSR LOPAGE       SET PAGE POINTERS TO LOWEST
2360                 ;                       PAGES IN ORIGIN, DESTINATION
                     ;                       BLOCKS.
2370                 ;
2380                 ;
2390 1822 A000              LDY #0           INITIALIZE PAGE INDEX TO
2400                 ;                       BOTTOM OF PAGE.
2410                 ;
2420 1824 AEB117            LDX NUM+1        USE X TO COUNT THE NUMBER
                     ;                       OF PAGES TO MOVE.  MORE THAN
                     ;                       ONE PAGE TO MOVE?
2430 1827 F00E              BEQ LESSDN       IF NOT, MOVE LESS THAN A
2440                 ;                       PAGE.
2450                 ;
2460                 ;                       IF SO,
2470 1829 B100      PAGEDN  LDA (GETPTR),Y   MOVE A PAGE DOWN,
2480 182B 9102              STA (PUTPTR),Y   STARTING AT THE BOTTOM.
2490 182D C8                INY              INCREMENT PAGE INDEX.
2500 182E D0F9              BNE PAGEDN       IF PAGE NOT MOVED, MOVE
2510                 ;                       NEXT BYTE...
2520                 ;
2530 1830 E601              INC GETPTR+1     INCREMENT PAGE POINTERS.
2540 1832 E603              INC PUTPTR+1
2550 1834 CA                DEX              DECREMENT PAGE COUNT.
2560 1835 D0F2              BNE PAGEDN       IF A PAGE LEFT TO MOVE,
2570                 ;                       MOVE IT AS A PAGE.
2580                 ;
2590 1837 88        LESSDN  DEY
2600 1838 C8                INY              MOVE LESS THAN A PAGE
2610 1839 B100              LDA (GETPTR),Y   DOWN, STARTING AT THE
2620 183B 9102              STA (PUTPTR),Y   BOTTOM.
2630 183D CCB017            CPY NUM          MOVED LAST BYTE?
2640 1840 D0F6              BNE LESSDN+1     IF NOT, MOVE NEXT BYTE...
2650 1842 4C1118            JMP OK.RTN       IF SO, RETURN BEARING
2660                 ;                       "OKAY" CODE.
2670                 ;
2680                 ;
2690                 ;
2700 1845 ADB117    MOVEUP  LDA NUM+1        MORE THAN A PAGE TO MOVE?
2710 1848 F048              BEQ LESSUP       IF NOT, MOVE LESS THAN A
2720                 ;                       PAGE.
```

```
2730                ;
2740                ;
2750                ;
2760                ;                          TO MOVE MORE THAN A PAGE,
2770                ;                          SET PAGE POINTERS TO
2780                ;                          HIGHEST PAGES IN ORIGIN,
2790                ;                          DESTINATION BLOCKS.
2800                ;
2810                ;
2820                ;                          TO DO THIS, FIRST
2830                ;                          SET (X,Y) = NUM - $FF,
2840                ;                          (RELATIVE ADDRESS OF
2850                ;                          HIGHEST PAGE IN A BLOCK.)
2860                ;
2870                ;
2880 184A ACB117            LDY NUM+1
2890 184D ADB017            LDA NUM
2900 1850 38                SEC
2910 1851 E9FF              SBC #$FF
2920 1853 B001              BCS NEXT.1
2930 1855 88                DEY
2940 1856 AA        NEXT.1  TAX
2950                ;
2960                ;                          NOW (X,Y) - NUM - $FF.
2970                ;                          X IS LOW BYTE, Y IS HIGH BYTE
2980                ;
2990                ;
3000 1857 8403              STY PUTPTR+1
3010 1859 8A                TXA
3020 185A 18                CLC
3030 185B 6D5215            ADC SA
3040 185E 8500              STA GETPTR
3050 1860 9001              BCC NEXT.2
3060 1862 C8                INY
3070                ;
3080                ;
3090 1863 98        NEXT.2  TYA
3100 1864 6D5315            ADC SA+1
3110 1867 8501              STA GETPTR+1
3120                ;
3130                ;    PTR=SA+NUM-$FF.
3140                ;                          (LAST PAGE IN SOURCE BLOCK.)
3150                ;
3160                ;
3170 1869 8A                TXA
3180 186A 18                CLC
3190 186B 6DB217            ADC DEST
3200 186E 8502              STA PUTPTR
3210 1870 9002              BCC NEXT.3
3220 1872 E603              INC PUTPTR+1
3230                ;
3240                ;
3250 1874 A503      NEXT.3  LDA PUTPTR+1
3260 1876 6DB317            ADC DEST+1
3270 1879 8503              STA PUTPTR+1
3280                ;                          NOW PUTPTR=DEST+NUM-$FF.
3290                ;                          (LAST PAGE IN DEST BLOCK.)
3300                ;
```

```
                 ;
                 ;
1878 AEB117           LDX NUM+1      LOAD X WITH NUMBER OF
                 ;                   PAGES TO MOVE.
                 ;
187E A0FF        PAGEUP LDY #$FF     SET PAGE INDEX TO TOP OF
                 ;                   PAGE.
1880 B100        LOOP.3 LDA (GETPTR),Y MOVE A PAGE UP, STARTING
1882 9102             STA (PUTPTR),Y AT THE TOP OF THE BLOCK.
1884 88               DEY            DECREMENT PAGE INDEX.
                 ;                   ABOUT TO MOVE LAST BYTE
                 ;                   IN PAGE?
1885 D0F9             BNE LOOP.3     IF NOT, HANDLE NEXT BYTE.
                 ;                   AS BEFORE.
                 ;
                 ;
                 ;
1887 B100             LDA (GETPTR),Y IF SO, MOVE THIS BYTE FROM
1889 9102             STA (PUTPTR),Y SOURCE TO DESTINATION.
188B C601             DEC GETPTR+1
188D C603             DEC PUTPTR+1   DECREMENT PAGE POINTERS.
188F CA               DEX            DECREMENT PAGE COUNTER.
1890 D0EC             BNE PAGEUP     IF A PAGE LEFT TO MOVE,
                 ;                   MOVE IT AS A PAGE....
                 ;
                 ;
1892 20A418      LESSUP JSR LOPAGE   MOVE LESS THAN A PAGE UP,
1895 ACB017           LDY NUM        STARTING AT THE TOP.
                 ;
1898 B100        MOVE.6 LDA (GETPTR),Y COPY A BYTE FROM ORIGIN
189A 9102             STA (PUTPTR),Y TO DESTINATION.
189C 88               DEY            DECREMENT PAGE INDEX.
189D C0FF             CPY #$FF       COPIED THE LAST BYTE?
189F D0F7             BNE MOVE.6     IF NOT, HANDLE AS BEFORE...
18A1 4C1118           JMP OK.RTN     IF SO, RETURN BEARING
                 ;                   "OKAY" CODE.
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;     SET PAGE POINTERS TO BOTTOM OF
                 ;       ORIGIN, DESTINATION BLOCKS.
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
```

```
3900 18A4 AD5215 LOPAGE LDA SA
3910 18A7 8500          STA GETPTR
3920 18A9 AD5315        LDA SA+1
3930 18AC 8501          STA GETPTR+1
3940                ;
3950                ;
3960 18AE ADB217        LDA DEST
3970 18B1 8502          STA PUTPTR
3980 18B3 ADB317        LDA DEST+1
3990 18B6 8503          STA PUTPTR+1
4000                ;
4010                ;
4020 18B8 60            RTS
4030                ;
4040                ;
4050                ;
4060                ;
4070                ;
4080                ;
4090                ;
4100                ;
4110                ; ********************************************
4120                ;
4130                ;     LET USER SET DESTINATION ADDRESS
4140                ;
4150                ; ********************************************
4160                ;
4170                ;
4180                ;
4190                ;
4200                ;
4210                ;
4220                ;
4230                ;
4240                ;
4250                ;
4260                ;
4270                ;
4280                ;
4290 18B9 200814 SET.DA JSR TVT.ON      LET USER SET DESTINATION
4300 18BC 20E414        JSR PRINT:
4310 18BF 7F            .BYTE TEX,CR,LF
4310 18C0 0D
4310 18C1 0A
4320 18C2 53             BYTE 'SET DESTINATION AND PRESS Q.'
4320 18C3 45
4320 18C4 54
4320 18C5 20
4320 18C6 44
4320 18C7 45
4320 18C8 53
4320 18C9 54
4320 18CA 49
4320 18CB 4E
4320 18CC 41
4320 18CD 54
4320 18CE 49
4320 18CF 4F
```

```
4320 18D0 4E
4320 18D1 20
4320 18D2 41
4320 18D3 4E
4320 18D4 44
4320 18D5 20
4320 18D6 50
4320 18D7 52
4320 18D8 45
4320 18D9 53
4320 18DA 53
4320 18DB 20
4320 18DC 51
4320 18DD 2E
4330 18DE FF            .BYTE ETX
4340 18DF 200712        JSR VISMON     LET USER SET AN ADDRESS.
4350 18E2 AD0512 DAHERE LDA SELECT     SET DEST=SELECT.
4360 18E5 8DB217        STA DEST
4370 18E8 AD0612        LDA SELECT+1
4380 18EB 8DB317        STA DEST+1
4390                 ;
4400 18EE 60            RTS            RETURN WITH DEST=SELECT.
```

Appendix C10:

Simple Text Editor (Top Level and Display Subroutines)

```
                ;       APPENDIX C10: ASSEMBLER LISTING OF
                ;            A SIMPLE TEXT EDITOR
                ;           TOP LEVEL AND DISPLAY SUBROUTINES
                ;
                ;
                ;
                ;
                ;
                ;     SEE CHAPTER 11 OF BEYOND GAMES: SYSTEMS
                ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                ;
                ;
                ;                    BY KEN SKIER
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;         CONSTANTS
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;
000D=               CR = $0D        CARRIAGE RETURN.
                ;
000A=               LF = $0A        LINE FEED.
                ;
                ;
007F=               TEX = $7F       THIS CHARACTER MUST START
                ;                   ANY MESSAGE.
                ;
00FF=               ETX = $FF       THIS CHARACTER MUST END
                ;                   ANY MESSAGE.
                ;
0049=               INSCHR='I       GRAPHIC FOR INSERT MODE
004F=               OVRCHR='O       GRAPHIC FOR OVERSTRIKE MODE.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;         EXTERNAL ADDRESSES
```

```
                ;
                ; ***********************************************
                ;
                ;
                ;
0000=                 TV.PTR=0       POINTER TO A SCREEN ADDRESS.
1000=                 PARAMS=$1000   SYSTEM DATA BLOCK.
                ;
                ;
1003=                 TVCOLS=PARAMS+3
1004=                 TVROWS=PARAMS+4
1007=                 ARROW=PARAMS+7
                ;
                ;
                ;
1100=                 TVSUBS=$1100
1113=                 CLR.XY=TVSUBS+$13
112B=                 TVHOME=TVSUBS+$2B
113C=                 TVTOXY=TVSUBS+$3C
1176=                 TVDOWN=TVSUBS+$76
117F=                 TVSKIP=TVSUBS+$7F
1181=                 TVPLUS=TVSUBS+$81
119B=                 TV.PUT=TVSUBS+$9B
11A3=                 VUBYTE=TVSUBS+$A3
11C4=                 TVPUSH=TVSUBS+$C4
11D3=                 TV.POP=TVSUBS+$D3
                ;
                ;
1200=                 VMPAGE=$1200   STARTING PAGE OF VISIBLE
                ;                    MONITOR CODE.
1205=                 SELECT=VMPAGE+5
1294=                 GET.SL=VMPAGE+$94
130D=                 INC.SL=VMPAGE+$10D
131A=                 DEC.SL=VMPAGE+$11A
                ;
                ;
1400=                 PRPAGE=$1400   STARTING PAGE OF PRINT
                ;                    UTILITIES.
1408=                 TVT.ON=PRPAGE+8
140E=                 TVTOFF=PRPAGE+$0E
1414=                 PR.ON =PRPAGE+$14
141A=                 PR.OFF=PRPAGE+$1A
1440=                 PR.CHR=PRPAGE+$40
14E4=                 PRINT:=PRPAGE+$E4
1512=                 PUSHSL=PRPAGE+$112
152B=                 POP.SL=PRPAGE+$12B
                ;
                ;
1500=                 HEX.PG=$1500   ADDRESS OF PAGE IN WHICH
                ;                    HEXDUMP CODE STARTS.
                ;
1552=                 SA=HEX.PG+$52
1554=                 EA=SA+2
15E9=                 SETADS=HEX.PG+$E9
1783=                 NEXTSL=HEX.PG+$283
17A0=                 GOTOSA=HEX.PG+$2A0
                ;
                ;
```

```
1170 1E00=            EDPAGE=$1E00   STARTING PAGE OF EDITOR.
1180 1EC8=            EDITIT=EDPAGE+$C8
1190             ;
1200             ;
1210             ;
1220             ;
1230             ;
1240             ;
1250             ; ********************************************
1260             ;
1270             ;         VARIABLES
1280             ;
1290             ; ********************************************
1300             ;
1310             ;
1320             ;
1330 1E00              *=EDPAGE
1340             ;
1350             ;
1360             ;
1370 1E00 00     COUNTR .BYTE 0        COUNTER USED BY LINE.2.
1380 1E01 00     EDMODE .BYTE 0        FLAG: 0=OVERSTRIKE,
1390             ;                           1=INSERT.
1400             ;
1410             ;
1420             ;
1430             ; ***********************************************
1440             ;
1450             ;         TEXT EDITOR: TOP LEVEL
1460             ;
1470             ; ***********************************************
1480             ;
1490             ;
1500             ;
1510             ;
1520             ;
1530             ;
1540 1E02 200F1E EDITOR JSR SETBUF     INITIALIZE BUFFER POINTERS.
1550 1E05 20371E EDLOOP JSR SHOWIT     SHOW USER A PORTION OF
1560             ;                     EDIT BUFFER.
1570 1E08 20C81E        JSR EDITIT     LET THE USER EDIT THE BUFFER
1580             ;                     OR MOVE ABOUT WITHIN IT.
1590 1E0B 18            CLC
1600 1E0C 18            CLC            LOOP BACK TO SHOW THE
1610 1E0D 90F6          BCC EDLOOP     CURRENT TEXT.
1620             ;
1630             ;
1640             ;
1650             ;
1660             ;
1670             ;
1680             ;
1690             ;
1700             ;
1710             ;
1720             ; ********************************************
1730             ;
1740             ;          INITIALIZE BUFFER POINTERS
```

```
1750                 ;
1760                 ; *********************************************
1770                 ;
1780                 ;
1790                 ;
1800                 ;
1810 1E0F 200814 SETBUF JSR TVT.ON     SELECT SCREEN.
1820 1E12 20E414        JSR PRINT:     DISPLAY "SET UP EDIT BUFFER."
1830 1E15 7F            .BYTE TEX,CR,LF,LF
1830 1E16 0D
1830 1E17 0A
1830 1E18 0A
1840 1E19 53            .BYTE 'SET UP EDIT BUFFER.
1840 1E1A 45
1840 1E1B 54
1840 1E1C 20
1840 1E1D 55
1840 1E1E 50
1840 1E1F 20
1840 1E20 45
1840 1E21 44
1840 1E22 49
1840 1E23 54
1840 1E24 20
1840 1E25 42
1840 1E26 55
1840 1E27 46
1840 1E28 46
1840 1E29 45
1840 1E2A 52
1840 1E2B 2E
1850 1E2C 0D            .BYTE CR,LF,LF,ETX
1850 1E2D 0A
1850 1E2E 0A
1850 1E2F FF
1860 1E30 20E915        JSR SETADS     LET USER SET LOCATION AND
1870                 ;                 SIZE OF EDIT BUFFER.
1880 1E33 20A017        JSR GOTOSA     SET SELECT=START OF BUFFER.
1890 1E36 60            RTS            RETURN TO CALLER.
1900                 ;
1910                 ;
1920                 ;
1930                 ;
1940                 ;
1950                 ;
1960                 ;
1970                 ; *********************************************
1980                 ;
1990                 ;      DISPLAY A PORTION OF EDIT BUFFER
2000                 ;
2020                 ; *********************************************
2030                 ;
2040                 ;
2050                 ;
2060                 ;
2070                 ;
2080 1E37 20C411 SHOWIT JSR TVPUSH     SAVE THE ZERO PAGE BYTES
2090                 ;                 WE'LL USE.
```

```
1E3A 202B11          JSR TVHOME     SET HOME POSITION OF EDIT
                ;                   DISPLAY.
                ;
                ;
1E3D AE0310          LDX TVCOLS     CLEAR THREE ROWS FOR
1E40 A003            LDY #3         THE EDIT DISPLAY.
1E42 201311          JSR CLR.XY
                ;
                ;
1E45 202B11          JSR TVHOME     RESTORE TV.PTR TO HOME
                ;                   POSITION OF EDIT DISPLAY.
1E48 207611          JSR TVDOWN     SET TV.PTR TO BEGINNING
1E4B 20C411          JSR TVPUSH     OF LINE TWO AND SAVE IT.
1E4E 205E1E          JSR LINE.2     DISPLAY TEXT IN LINE TWO.
                ;
                ;
1E51 20D311          JSR TV.POP     SET TV.PTR TO BEGINNING OF
1E54 207611          JSR TVDOWN     OF THIRD LINE OF EDIT
                ;                   DISPLAY.
1E57 20891E          JSR LINE.3     DISPLAY THIRD LINE OF EDIT
                ;                   DISPLAY.
                ;
1E5A 20D311          JSR TV.POP     RESTORE ZERO PAGE BYTES USED.
1E5D 60              RTS            RETURN TO CALLER, WITH EDIT
                ;                   DISPLAY ON SCREEN, REST OF
                ;                   SCREEN UNCHANGED, AND ZERO
                ;                   PAGE PRESERVED.
                ;
                ;
                ;
                ;
                ;
                ;
                ; *********************************************
                ;
                ;          DISPLAY TEXT LINE
                ;
                ; *********************************************
                ;
                ;
                ;
                ;
                ;
1E5E 201215 LINE.2 JSR PUSHSL       SAVE SELECT POINTER.
1E61 AD0310          LDA TVCOLS     SET X EQUAL TO
1E64 4A              LSR A          HALF THE WIDTH
1E65 AA              TAX            OF THE SCREEN.
1E66 CA              DEX
1E67 CA              DEX
                ;
1E68 201A13 LOOP.1 JSR DEC.SL       DECREMENT SELECT...
1E6B CA              DEX
1E6C 10FA            BPL LOOP.1     ...X TIMES.
                ;
1E6E AD0310          LDA TVCOLS     INITIALIZE COUNTR.
1E71 8D001E          STA COUNTR     (WE'LL DISPLAY TVCOLS
                ;                   CHARACTERS.)
1E74 209412 LOOP.2 JSR GET.SL       GET A CHARACTER FROM BUFFER.
```

```
1E77 209B11          JSR TV.PUT      PUT IT ON SCREEN.
1E7A 207F11          JSR TVSKIP      GO TO NEXT SCREEN POSITION.
1E7D 200D13          JSR INC.SL      ADVANCE TO NEXT BYTE IN
                ;                    BUFFER.
1E80 CE001E          DEC COUNTR      DONE LAST CHARACTER IN ROW?
1E83 10EF            BPL LOOP.2      IF NOT, DO NEXT CHARACTER.
                ;
                ;
1E85 202B15          JSR POP.SL      RESTORE SELECT FROM STACK.
1E88 60              RTS             RETURN TO CALLER.
                ;
                ;
                ;
                ;
                ;
                ; *********************************************
                ;
                ;         DISPLAY STATUS LINE
                ;
                ; *********************************************
                ;
                ;
                ;
                ;
                ;
1E89 AD0310 LINE.3 LDA TVCOLS        SELECT CENTER POSITION...
1E8C 4A              LSR A           A=TVCOLS/2
1E8D E902            SBC #2          A=(TVCOLS/2)-2
1E8F 208111          JSR TVPLUS      NOW TV.PTR IS POINTING TWO
                ;                    CHARACTERS TO THE LEFT OF
                ;                    CENTER OF LINE 3 OF THE
                ;                    EDIT DISPLAY.
1E92 AD011E          LDA EDMODE      WHAT IS CURRENT MODE?
1E95 C901            CMP #1          IS IT INSERT MODE?
1E97 D005            BNE OVMODE      IF NOT, IT MUST BE OVERSTRIKE
                ;                    MODE.
1E99 A949            LDA #INSCHR     IF SO, GET INSERT GRAPHIC.
1E9B 18              CLC
1E9C 9002            BCC TVMODE
1E9E A94F   OVMODE LDA #OVRCHR       LOAD A W/OVERSTRIKE CHARACTER.
1EA0 209B11 TVMODE JSR TV.PUT        PUT MODE GRAPHIC ON SCREEN.
1EA3 A902            LDA #2          MOVE TWO POSITIONS TO THE
1EA5 208111          JSR TVPLUS      RIGHT, SO TV.PTR POINTS TO
                ;                    CENTER OF LINE 3 OF EDIT
                ;                    DISPLAY.
1EA8 AD0710          LDA ARROW       DISPLAY AN UP-ARROW HERE.
1EAB 209B11          JSR TV.PUT
                ;
1EAE A902            LDA #2          GO TWO POSITIONS TO THE
1EB0 208111          JSR TVPLUS      RIGHT, SO TV.PTR POINTS TO
                ;                    FIELD RESERVED FOR THE
                ;                    ADDRESS OF THE CURRENT CHARACTER
1EB3 AD0612          LDA SELECT+1    DISPLAY ADDRESS OF CURRENT
1EB6 20A311          JSR VUBYTE
1EB9 AD0512          LDA SELECT
1EBC 20A311          JSR VUBYTE
                ;
1EBF 60              RTS             RETURN TO CALLER.
```

Appendix C11:

Simple Text Editor (EDITIT Subroutine)

```
                ;       APPENDIX C11: ASSEMBLER LISTING OF
                ;            A SIMPLE TEXT EDITOR
                ;          EDITIT SUBROUTINE
                ;
                ;
                ;
                ;
                ;
                ;     SEE CHAPTER 11 OF BEYOND GAMES: SYSTEMS
                ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                ;
                ;
                ;                  BY KEN SKIER
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;          CONSTANTS
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;
000D=                CR = $0D        CARRIAGE RETURN.
                ;
000A=                LF = $0A        LINE FEED.
                ;
                ;
007F=                TEX = $7F       THIS CHARACTER MUST START
                ;                    ANY MESSAGE.
                ;
00FF=                ETX = $FF       THIS CHARACTER MUST END
                ;                    ANY MESSAGE.
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;          EXTERNAL ADDRESSES
                ;
```

```
580                 ; **********************************************
590                 ;
600                 ;
610                 ;
620                 ;
630                 ;
640 1200=                VMPAGE=$1200   STARTING PAGE OF VISIBLE
650                 ;                   MONITOR CODE.
660 1205=                SELECT=VMPAGE+5
670 1207=                VISMON=VMPAGE+7
680 1294=                GET.SL=VMPAGE+$94
690 12E0=                GETKEY=VMPAGE+$E0
700 130D=                INC.SL=VMPAGE+$10D
710 131A=                DEC.SL=VMPAGE+$11A
720 132D=                PUT.SL=VMPAGE+$12D
730                 ;
740                 ;
750 1400=                PRPAGE=$1400   STARTING PAGE OF PRINT
760                 ;                   UTILITIES.
765 1414=                PR.ON =PRPAGE+$14
767 141A=                PR.OFF=PRPAGE+$1A
770 1440=                PR.CHR=PRPAGE+$40
780 14E4=                PRINT:=PRPAGE+$E4
790 1512=                PUSHSL=PRPAGE+$112
800 152B=                POP.SL=PRPAGE+$12B
810                 ;
820                 ;
830 1500=                HEX.PG=$1500   ADDRESS OF PAGE IN WHICH
840                 ;                   HEXDUMP CODE STARTS.
850                 ;
860 1552=                SA=HEX.PG+$52
870 1554=                EA=SA+2
880 1667=                SAHERE=HEX.PG+$167
890 1783=                NEXTSL=HEX.PG+$283
900 17A0=                GOTOSA=HEX.PG+$2A0
910                 ;
920                 ;
930 17B0=                MOVERS=$17B0   START OF MOVE OBJECT CODE.
940 17B2=                DEST   =MOVERS+2
950 17D6=                MOV.EA=MOVERS+$26
960 18E2=                DAHERE=MOVERS+$132
970                 ;
980 1E00=                EDPAGE=$1E00   STARTING PAGE OF EDITOR.
990 1EC0=                EDKEYS=EDPAGE+$C0
1000                ;
1010                ;
1020                ;
1030                ;
1040                ;
1050                ;
1060                ; **********************************************
1070                ;
1080                ;        VARIABLES
1090                ;
1100                ; **********************************************
1110                ;
1120                ;
1130                ;
```

```
1E00                 *=EDPAGE
               ;
               ;
               ;
1E01=                EDMODE=EDPAGE+1       0=OVERSTRIKE MODE.
               ;                           1=INSERT.
               ;
1EC0                 *=EDKEYS
               ;
               ;          EDIT FUNCTION KEYS
               ;
               ;                      THE EDITOR RECOGNIZES THE
               ;                      FOLLOWING KEYS AS FUNCTION KEYS.
               ;                      ASSIGN A FUNCTION TO A KEY
               ;                      BY STORING THE DESIRED KEY
               ;                      CODE FROM YOUR SYSTEM'S
               ;                      KEYHANDLER INTO ONE OF THE
               ;                      FOLLOWING DATA BYTES:
               ;
               ;
1EC0 06       FLSHKY .BYTE $06        THIS KEY FLUSHES THE
               ;                      BUFFER OF ANY TEXT.  $06 IS
               ;                      CONTROL-F.  THUS, CONTROL-F
               ;                      TO FLUSH THE BUFFER.
               ;
               ;
1EC1 03       MODEKY .BYTE $03        THIS KEY CAUSES THE EDIT
               ;                      TO CHANGE MODES. FROM INSERT
               ;                      TO OVERSTRIKE, AND VICE VERSA.
               ;                      $03 IS CONTROL-C.  THUS,
               ;                      CONTROL-C TO Change modes.
               ;
1EC2 3E       NEXTKY .BYTE '>'        THIS KEY SELECTS THE NEXT
               ;                      CHARACTER IN THE BUFFER.
               ;
               ;                      SUBSTITUTE RIGHT-ARROW IF
               ;                      YOUR KEYBOARD HAS IT.
               ;
1EC3 3C       PREVKY .BYTE '<'        SELECT PREVIOUS CHARACTER
               ;                      IN THE BUFFER.  SUBSTITUTE
               ;                      LEFT-ARROW IF YOUR KEYBOARD
               ;                      HAS IT.
               ;
1EC4 10       PRTKEY .BYTE $10        THIS KEY PRINTS THE BUFFER.
               ;                      CONTROL-P
               ;                      to Print the buffer.
               ;
1EC5 7F       RUBKEY .BYTE $7F        THIS KEY RUBS OUT THE
               ;                      CURRENT CHARACTER.  IF YOU
               ;                      HAVE DELETE KEY BUT NOT RUBOUT,
               ;                      USE YOUR SYSTEM'S CODE FOR
               ;                      THE DELETE KEY.
               ;
               ;
1EC6 51       QUITKY .BYTE 'Q'        TWO QUIT KEYS IN A ROW
               ;                      CAUSE THE EDITOR TO RETURN
               ;                      TO ITS CALLER.
               ;
```

```
1720                ;
1730                ;
1740                ;
1750                ;
1760                ;                         OTHER VARIABLES:
1770                ;
1780 1EC7 00       TEMPCH .BYTE 0             THIS BYTE USED BY EDITIT.
1790                ;
1800                ;
1810                ;
1820                ;
1830                ;
1840                ;
1850                ;
1860                ;
1870                ;
1880                ;
1890                ; ***********************************************
1900                ;
1910                ;         TEXT EDITOR: UPDATE SUBROUTINE
1920                ;
1930                ; ***********************************************
1940                ;
1950                ;
1960                ;
1970                ;
1980                ;
1990                ;
2000                ;
2010 1EC8 20E012 EDITIT JSR GETKEY            GET A KEYSTROKE FROM USER
2020                ;                         USER.
2030 1ECB CDC61E        CMP QUITKY            IS IT THE "QUIT" KEY?
2040 1ECE D017          BNE DO.KEY            IF NOT, DO WHAT THE KEY
2050                ;                         REQUIRES.
2060                ;
2070 1ED0 48            PHA                   IF IT IS THE "QUIT" KEY, SAVE
2080 1ED1 20E012        JSR GETKEY            IT AND GET A NEW KEY FROM
2090                ;                         USER.
2100 1ED4 CDC61E        CMP QUITKY            IS THIS A "QUIT" KEY, TOO?
2110 1ED7 D004          BNE NOTEND            IF NOT, THEN THIS IS NOT THE
2120                ;                         END OF THE EDIT SESSION.
2130                ;
2140                ;                         END THE EDT SESSION?
2150 1ED9 68       ENDEDT PLA                 POP FIRST "QUIT" KEY FROM
2160                ;                         STACK.
2170 1EDA 68            PLA                   POP RETURN ADDRESS TO
2180 1EDB 68            PLA                   EDITOR'S TOP LEVEL.
2190 1EDC 60            RTS                   RETURN TO EDITOR'S CALLER.
2200                ;
2210 1EDD 8DC71E NOTEND STA TEMPCH            SAVE TH KEY THAT FOLLOWED
2220                ;                         THE "QUIT" KEY.
2230 1EE0 68            PLA                   POP FIRST "QUIT" KEY FROM STACK.
2240 1EE1 20E71E        JSR DO.KEY            DO WHAT IT REQUIRES.
2250 1EE4 ADC71E        LDA TEMPCH            RECOVER THE KEY THAT FOLLOWED
2260                ;                         THE "QUIT" KEY.
2270                ;
2280                ;                         "DO.KEY" DOES WHAT THE KEY
2290                ;                         IN THE ACCUMULATOR REQUIRES:
```

```
2300                  ;
2310 1EE7 CDC11E DO.KEY CMP MODEKY     IS IT THE "CHANGE MODE" KEY?
2320 1EEA D00B          BNE IFNEXT     IF NOT, PERFORM NEXT TEST.
2330 1EEC CE011E        DEC EDMODE     IF SO, CHANGE THE EDITOR'S
2340 1EEF 1005          BPL DO.END     MODE.
2350 1EF1 A901          LDA #1
2360 1EF3 8D011E        STA EDMODE
2370 1EF6 60     DO.END RTS            RETURN TO CALLER.
2380                  ;
2390                  ;
2400 1EF7 CDC21E IFNEXT CMP NEXTKY     IS IT THE "NEXT" KEY?
2410 1EFA D004          BNE IFPREV     IF NOT, PERFORM NEXT TEST.
2420                  ;
2430 1EFC 20791F        JSR NEXTCH     IF SO, ADVANCE TO NEXT
2440                  ;                CHARACTER...
2450 1EFF 60            RTS            ...AND RETURN.
2460                  ;
2470                  ;
2480 1F00 CDC31E IFPREV CMP PREVKY     IS IT THE "PREVIOUS" KEY?
2490 1F03 D004          BNE IF.RUB     IF NOT, PERFORM NEXT TEST.
2500 1F05 20871F        JSR PREVSL     IF SO, BACK UP TO PREVIOUS
2510 1F08 60            RTS            CHARACTER AND RETURN.
2520                  ;
2530                  ;
2540 1F09 CDC51E IF.RUB CMP RUBKEY     IS IT THE "RUBOUT" KEY?
2550 1F0C D004          BNE IF.PRT     IF NOT, PERFORM NEXT TEST.
2560 1F0E 20DD1F        JSR DELETE     IF SO, DELETE CURRENT
2570 1F11 60            RTS            CHARACTER AND RETURN.
2580                  ;
2590                  ;
2600 1F12 CDC41E IF.PRT CMP PRTKEY     IS IT THE "PRINT" KEY?
2610 1F15 D004          BNE IFFLSH     IF NOT, PERFORM NEXT TEST.
2620 1F17 20C51F        JSR PRTBUF     IF SO, PRINT THE BUFFER...
2630 1F1A 60            RTS            ...AND RETURN.
2640                  ;
2650                  ;
2660                  ;
2670 1F1B CDC01E IFFLSH CMP FLSHKY     IS IT THE "FLUSH" KEY?
2680 1F1E D004          BNE CHARKY     IF NOT, IT MUST BE A CHARACTER
2690                  ;                KEY.
2700 1F20 20B41F        JSR FLUSH      IF SO, FLUSH THE BUFFER.
2710 1F23 60            RTS            AND RETURN.
2720                  ;
2730                  ;
2740                  ;
2750                  ;
2760                  ;    OK.  IT'S NOT AN EDITOR FUNCTION KEY, SO IT
2770                  ; MUST BE A CHARACTER KEY.  DEPENDING ON THE
2780                  ; CURRENT MODE, WE'LL EITHER INSERT OR OVERSTRIKE
2790                  ;                THE CURRENT CHARACTER.
2800                  ;
2810 1F24 AE011E CHARKY LDX EDMODE     ARE WE IN OVERSTRIKE MODE?
2820 1F27 F004          BEQ STRIKE     IF SO, OVERSTRIKE THE CURRENT
2830                  ;                CHARACTER.
2840 1F29 20341F        JSR INSERT     IF NOT, INSERT THE CHARACTER.
2850 1F2C 60            RTS            RETURN.
2860                  ;
2870 1F2D 202D13 STRIKE JSR PUT.SL     REPLACE CURRENT CHARACTER
```

```
                ;                  WITH NEW CHARACTER.
1F30 208317             JSR NEXTSL   SELECT NEXT CHARACTER.
1F33 60                 RTS          RETURN.
                ;
                ;
                ;
                ;
                ;
1F34 48         INSERT  PHA          SAVE THE CHARACTER TO BE
                ;                    INSERTED, WHILE WE MAKE ROOM
                ;                    FOR IT IN THE BUFFER...
1F35 201215             JSR PUSHSL   SAVE THE CURRENT ADDRESS.
1F38 AD5315             LDA SA+1     SAVE THE BUFFER'S ADDRESS.
1F3B 48                 PHA
1F3C AD5215             LDA SA
1F3F 48                 PHA
                ;
                ;
1F40 AD5515             LDA EA+1     SAVE BUFFER'S END ADDRESS.
1F43 48                 PHA
1F44 AD5415             LDA EA
1F47 48                 PHA
                ;
                ;
1F48 206716             JSR SAHERE   SET SA=SELECT, SO CURRENT
                ;                    LOCATION WILL BE START OF
                ;                    THE BLOCK WE'LL MOVE.
                ;
                ;
                ;
1F4B 208317             JSR NEXTSL   ADVANCE TO NEXT CHARACTER
                ;                    POSITION IN THE BUFFER.
1F4E 3011               BMI ENDINS   IF WE'RE AT THE END OF THE
                ;                    BUFFER, WE'LL OVERSTRIKE
                ;                    INSTEAD OF INSERTING.
                ;
                ;
1F50 20E218             JSR DAHERE   SET DEST=SELECT.
                ;                    DESTINATION OF BLOCK MOVE
                ;                    WILL BE ONE BYTE ABOVE
                ;                    BLOCK'S INITIAL LOCATION.
                ;
                ;
1F53 AD5415             LDA EA       DECREMENT END ADDRESS
1F56 D004               BNE *+6
1F58 CE5515             DEC EA+1
1F5B CE5415             DEC EA
                ;
                ;
                ;
1F5E 20D617     OPENUP  JSR MOV.EA   OPEN UP ONE BYTE OF SPACE
                ;                    AT CURRENT CHARACTER'S
                ;                    LOCATION, BY MOVING TO DEST
                ;                    THE BLOCK SPECIFIED BY SA, EA.
                ;
                ;
1F61 68         ENDINS  PLA          RESTORE EA SO IT POINTS
1F62 8D5415             STA EA       TO END OF BUFFER.
```

```
1F65 68              PLA
1F66 8D5515          STA EA+1
                ;
                ;
1F69 68              PLA            RESTORE SA SO IT POINTS TO
1F6A 8D5215          STA SA         START OF BUFFER.
1F6D 68              PLA
1F6E 8D5315          STA SA+1
                ;
                ;
1F71 202B15          JSR POP.SL     RESTORE SELECT SO IT POINTS
                ;                   TO CURRENT CHARACTER POSITION.
                ;
                ;
1F74 68              PLA            RESTORE NEW CHARACTER TO
                ;                   ACCUMULATOR.  WE'VE CREATED
                ;                   A ONE-BYTE SPACE FOR IT, SO
1F75 202D1F          JSR STRIKE     WE NEED ONLY OVERSTRIKE IT
1F78 60              RTS            AND RETURN.
1F79 209412 NEXTCH JSR GET.SL       GET CURRENT CHARACTER.
1F7C C9FF            CMP #ETX       IS IT END OF TEXT CHARACTER?
1F7E F004            BEQ AN.ETX     IF SO, RETURN TO CALLER,
                ;                   BEARING A NEGATIVE RETURN CODE.
                ;
1F80 208317          JSR NEXTSL     IF NOT, SELECT NEXT BYTE IN
                ;                   BUFFER.
1F83 60              RTS            RETURN PLUS IF WE INCREMENTED
                ;                   SELECT; MINUS IF SELECT
                ;                   ALREADY EQUALLED EA.
                ;
1F84 A9FF   AN.ETX LDA #$FF         SINCE WE'RE ON AN ETX, WE
1F86 60              RTS            WILL RETURN MINUS, WITHOUT
                ;                   INCREMENTING SELECT.
                ;
                ;
                ;
                ;
                ;
1F87 38     PREVSL SEC              PREPARE TO COMPARE.
1F88 AD5315          LDA SA+1       IS SELECT IN A HIGHER PAGE
1F8B CD0612          CMP SELECT+1   THAN START OF BUFFER?
1F8E 900C            BCC SL.OK      IF SO, SELECT MAY BE DECREMENTED
1F90 D010            BNE NOT.OK     IF SELECT IS IN A LOWER
                ;                   PAGE THAN SA, IT'S NOT OK.
                ;
                ;                   SELECT IS IN SAME PAGE AS SA.
1F92 AD5215          LDA SA         IS SELECT>SA?
1F95 CD0512          CMP SELECT
1F98 F017            BEQ NO.DEC     IF SELECT=SA, DON'T DECREMENT
                ;                   SELECT.
1F9A B006            BCS NOT.OK     IF SELECT<SA, DON'T DECREMENT
                ;                   SELECT.
1F9C 201A13 SL.OK  JSR DEC.SL       SELECT>SA, SO WE MAY
                ;                   DECREMENT SELECT AND IT
                ;                   WILL REMAIN IN THE BUFFER.
1F9F A900            LDA #0         SET A POSITIVE RETURN CODE...
1FA1 60              RTS            ...AND RETURN.
                ;
                ;
```

```
1FA2 AD5215 NOT.OK LDA SA          SINCE SELECT<SA, IT IS NOT
1FA5 8D0512        STA SELECT      EVEN IN THE EDIT BUFFER.  SO
1FA8 AD5315        LDA SA+1        MAKE SELECT LEGAL, BY SETTING
1FAB 8D0612        STA SELECT+1    IT EQUAL TO SA.
1FAE A900          LDA #0          SET A POSITIVE RETURN CODE...
1FB0 60            RTS             ...AND RETURN.
            ;
            ;
1FB1 A9FF   NO.DEC LDA #$FF        SELECT=SA, SO CHANGE
1FB3 60            RTS             NOTHING.  RETURN WITH
            ;                      NEGATIVE RTURN CODE.
            ;
            ;
            ;
1FB4 20A017 FLUSH  JSR GOTOSA      SET SELECT=SA.
1FB7 A9FF   FLOOP  LDA #ETX        PUT AN ETX CHARACTER
1FB9 202D13        JSR PUT.SL      INTO THE BUFFER.
1FBC 208317        JSR NEXTSL      ADVANCE TO NEXT POSITION IN
            ;                      BUFFER.
1FBF 10F6          BPL FLOOP       IF WE HAVEN'T REACHED END
            ;                      OF BUFFER, PUT AN ETX INTO
            ;                      THIS POSITION, TOO.
            ;
1FC1 20A017        JSR GOTOSA      HAVING FILLED BUFFER WITH
            ;                      ETC CHARACTERS, RESET SELECT
            ;                      TO BEGINNING OF BUFFER.
1FC4 60            RTS             RETURN.
1FC5 20A017 PRTBUF JSR GOTOSA      SET SELECT TO START OF BUFFER
1FC8 201414        JSR PR.ON       SELECT PRINTER FOR OUTPUT.
1FCB 209412 PRLOOP JSR GET.SL       GET CURRENT CHARACTER.
1FCE C9FF          CMP #ETX        IS IT ETX?
1FD0 F008          BEQ ENDPRT      IF SO, WE'RE DONE.
1FD2 204014        JSR PR.CHR      IF NOT, PRINT IT.
1FD5 208317        JSR NEXTSL      SELECT NEXT CHARACTER
1FD8 10F1          BPL PRLOOP      IF WE HAVEN'T REACHED THE
            ;                      END OF THE BUFFER, HANDLE
            ;                      THE CURRENT CHARACTER AS BEFORE.
1FDA 4C1A14 ENDPRT JMP PR.OFF      HAVING REACHED END OF MESSAGE
            ;                      OR END OF BUFFER, RETURN TO
            ;                      CALLER OF EDITIT, DESELECTING
            ;                      THE PRINTER AS WE DO SO.
            ;
            ;
1FDD 201215 DELETE JSR PUSHSL      SAVE CURRENT ADDRESS.
1FE0 AD5315        LDA SA+1        SAVE BUFFER'S START ADDRESS.
1FE3 48            PHA
1FE4 AD5215        LDA SA
1FE7 48            PHA
            ;
1FE8 20E218        JSR DAHERE      SET DEST=SELECT, BECAUSE
            ;                      WE'LL MOVE A BLOCK OF TEXT
            ;                      DOWN TO HERE, TO CLOSE UP
            ;                      THE BUFFER AT THE CURRENT
            ;                      CHARACTER.
1FEB 208317        JSR NEXTSL      ADVANCE BY ONE BYTE THROUGH
            ;                      BUFFER, IF POSSIBLE.
1FEE 206716        JSR SAHERE      SET SA=SELECT, BECAUSE THIS
            ;                      IS THE START OF THE BLOCK WE'LL
```

```
4620                  ;                  MOVE DOWN.
4630                  ;                  NOTE: THE ENDING ADDRESS OF
4640                  ;                  THE BLOCK IS THE END ADDRESS
4650                  ;                  OF THE TEXT BUFFER.
4660 1FF1 20D617           JSR MOV.EA    MOVE BLOCK SPECIFIED BY
4670                  ;                  SA, EA TO DEST.
4680                  ;
4690                  ;
4700 1FF4 68               PLA           RESTORE INITIAL SA (WHICH
4710 1FF5 8D5215           STA SA        IS THE START ADDRESS OF THE
4720 1FF8 68               PLA           TEXT BUFFER, NOT OF THE BLOCK
4730 1FF9 8D5315           STA SA+1      WE JUST MOVED.)
4740 1FFC 202B15           JSR POP.SL    RESTORE CURRENT ADDRESS.
4750 1FFF 60               RTS           RETURN TO CALLER.
```

Appendix C12:

Extending the Visible Monitor

```
          ;       APPENDIX C12: ASSEMBLER LISTING OF
          ;          VISIBLE MONITOR EXTENSIONS
          ;
          ;
          ;
          ;
          ;
          ;      SEE CHAPTER 12 OF BEYOND GAMES: SYSTEM
          ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ;
          ; ********************************************
          ;
          ;          EXTERNAL ADDRESSES
          ;
          ; ********************************************
          ;
          ;
          ;
          ;
          ;
          ;
1400=           PRPAGE=$1400   STARTING PAGE OF PRINT
          ;                    UTILITIES.
1400=           PRINTR=PRPAGE
1402=           USER  =PRPAGE+2
          ;
          ;
1500=           HEX.PG=$1500   ADDRESS OF PAGE IN WHICH
          ;                    HEXDUMP CODE STARTS.
          ;
1557=           TVDUMP=HEX.PG+$57
15AE=           PRDUMP=HEX.PG+$AE
          ;
          ;
1900=           DSPAGE=$1900   STARTING PAGE OF DISASSEMBLER
```

```
580 1909=              TV.DIS=DSPAGE+9
590 1926=              PR.DIS=DSPAGE+$26
600                ;
610 17B0=              MOVERS=$17B0   START OF MOVE OBJECT CODE.
620 17B4=              MOVER =MOVERS+4
630                ;
640                ;
650 1E00=              EDPAGE=$1E00   ADDRESS OF PAGE IN WHICH
660                ;                  EDITOR CODE BEGINS.
670 1E02=              EDITOR=EDPAGE+2
680                ;
690                ;
700                ;
710                ;
720                ;
730                ;
740                ;
750                ;
760 10B0               *=$10B0
770                ;
780                ;
790                ; ********************************************
800                ;
810                ;     EXTENSIONS TO THE VISIBLE MONITOR
820                ;
830                ; ********************************************
840                ;
850                ;
860                ;
870 10B0 C950   EXTEND CMP #'P        IS IT THE 'P' KEY?
880 10B2 D009          BNE IF.U       IF NOT, PERFORM NEXT TEST.
890 10B4 AD0014        LDA PRINTR     IF SO, TOGGLE THE PRINTER
900 10B7 49FF          EOR #$FF       FLAG...
910 10B9 8D0014        STA PRINTR
920 10BC 60            RTS            AND RETURN TO CALLER.
930                ;
940 10BD C955   IF.U   CMP #'U         IS IT THE 'U' KEY?
950 10BF D009          BNE IF.H       IF NOT, PERFORM NEXT TEST.
960 10C1 AD0214        LDA USER       IF SO, TOGGLE THE USER-
970 10C4 49FF          EOR #$FF       PROVIDED OUTPUT FLAG...
980 10C6 8D0214        STA USER
990 10C9 60            RTS            AND RETURN.
1000               ;
1010 10CA C948  IF.H   CMP #'H        IS IT THE 'H' KEY?
1020 10CC D00D         BNE IF.M       IF NOT, PERFORM NEXT TEST.
1030 10CE AD0014       LDA PRINTR     IS THE PRINTER SELECTED?
1040 10D1 D004         BNE NEXT.1     IF SO, PRINT A HEXDUMP.
1050 10D3 205715       JSR TVDUMP     IF NOT, DUMP TO SCREEN...
1060 10D6 60           RTS         AND RETURN.
1070 10D7 20AE15 NEXT.1 JSR PRDUMP    PRINT A HEXDUMP...
1080 10DA 60           RTS            ...AND RETURN.
1090               ;
1100 10DB C94D  IF.M   CMP #'M        IS IT THE 'M' KEY?
1110 10DD D004         BNE IF.DIS     IF NOT, PRFORM NEXT TEST.
1120 10DF 20B417       JSR MOVER      IF SO, LET USER SPECIFY AND
1130 10E2 60           RTS            AND MOVE A BLOCK OF MEMORY.
1140               ;
1150 10E3 C93F  IF.DIS CMP #'?        IS IT THE '?' KEY?
```

```
1160 10E5 D00D            BNE IF.T        IF NOT, PERFORM NEXT TEST.
1170 10E7 AD0014          LDA PRINTR      IS THE PRINTER SELECTED?
1160 10EA D004            BNE NEXT.2      IF SO, PRINT A DISASSEMBLY.
1190 10EC 200919          JSR TV.DIS      IF NOT, DISASSEMBLE TO THE
1200 10EF 60              RTS             SCREEN AND RETURN.
1210 10F0 202619 NEXT.2   JSR PR.DIS      PRINT A DISASSEMBLY...
1220 10F3 60              RTS             AND RETURN.
1230             ;
1240 10F4 C954   IF.T     CMP #'T         IS IT THE 'T' KEY?
1250 10F6 D004            BNE EXIT        IF NOT, RETURN.
1260 10F8 20021E          JSR EDITOR      IF SO, CALL THE SIMPLE
1270 10FB 60              RTS             TEXT EDITOR AND RETURN.
1280             ;
1290 10FC 60     EXIT     RTS             EXTEND THE VISIBLE MONITOR
1300             ;                        EVEN FURTHER BY REPLACING
1310             ;                        THIS 'RTS' WITH A 'JMP' TO
1320             ;                        MORE TEST-AND-BRANCH CODE.
```

Appendix C13:

System Data Block for the Ohio Scientific C-1P

```
                 ;          APPENDIX C13: ASSEMBLER LISTING OF
                 ;             SYSTEM DATA BLOCK
                 ;         FOR THE OHIO SCIENTIFIC C-1P
                 ;
                 ;
                 ;
                 ;
                 ;      SEE APPENDIX B1 OF BEYOND GAMES: SYSTEM
                 ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                 ;
                 ;
                 ;                    BY KEN SKIER
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ;
                 ; ********************************************
                 ;
                 ;          SCREEN PARAMETERS
                 ;
                 ; ********************************************
                 ;
                 ;
                 ;
                 ;
                 ;
1000                    *=$1000
                 ;
                 ;
                 ;
                 ;
                 ;
1000 65D0        HOME    .WORD $D065      THIS IS THE ADDRESS OF THE
                 ;                        CHARACTER IN THE UPPER LEFT
                 ;                        CORNER OF THE SCREEN.  THE
                 ;                        ADDRESS OF HOME WILL VARY AS
                 ;                        A FUNCTION OF YOUR VIDEO MONITOR
                 ;                        I SET MINE TO $D065.  IF YOU
                 ;                        CAN'T SEE THE VISIBLE MONITOR
                 ;                        DISPLAY, ADJUST THE LOW BYTE.
                 ;
                 ;
                 ;
1002 20          ROWINC .BYTE 32          ADDRESS DIFFERENCE FROM ONE
                 ;                        ROW TO THE NEXT.
1003 18          TVCOLS .BYTE $18         NUMBER OF COLUMNS ON SCREEN,
                 ;                        COUNTING FROM ZERO.
1004 18          TVROWS .BYTE $18         NUMBER OF ROWS ON SCREEN,
                 ;                        COUNTING FROM ZERO.
```

```
1005 D3      HIPAGE  .BYTE $D3      HIGHEST PAGE IN SCREEN MEMORY.
1006 20      BLANK   .BYTE $20      OSI DISPLAY CODE FOR A BLANK.
1007 10      ARROW   .BYTE $10      OSI DISPLAY CODE FOR AN UP-ARROW
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ;
              ; ************************************************
              ;
              ;               INPUT/OUTPUT VECTORS
              ;
              ; ************************************************
              ;
              ;
              ;
              ;
              ;
              ;
1008 EDFE    ROMKEY .WORD $FEED     POINTER TO ROUTINE THAT GETS
              ;                     AN ASCII CHARACTER FROM THE
              ;                     KEYBOARD.  (NOTE: $FFEB IS
              ;                     THE GENERAL CHARACTER-INPUT
              ;                     ROUTINE FOR OSI BASIC-IN-ROM
              ;                     COMPUTERS.)
              ;
              ;
100A 2DBF    ROMTVT .WORD $BF2D     POINTER TO ROUTINE TO PRINT
              ;                     AN ASCII CHARACTER ON THE SCREEN
              ;                     (NOTE: $FFEE IS THE
              ;                     CHARACTER-OUTPUT ROUTINE FOR
              ;                     OSI BASIC-IN-ROM COMPUTERS.)
              ;
              ;
100C B1FC    ROMPRT .WORD $FCB1     POINTER TO ROUTINE TO SEND AN
              ;                     ASCII CHARACTER TO THE PRINTER
              ;                     (ACTUALLY, TO THE CASSETTE PORT.
              ;
              ;
100E 1010    USROUT .WORD DUMMY     POINTER TO USER-WRITTEN OUTPUT
              ;                     ROUTINE.  (SET HERE TO DUMMY
              ;                     UNTIL YOU SET IT TO POINT
              ;                     TO YOUR OWN CHARACTER-OUTPUT
              ;                     ROUTINE.)
              ;
              ;
1010 60      DUMMY  RTS             THIS IS A DUMMY SUBROUTINE.
              ;                     IT DOES NOTHING BUT RETURN.
              ;
              ;
              ;
              ;
              ;
              ;
```

```
                ; ****************************************
                ;
                ;      CONVERT ASCII CHARACTER TO DISPLAY CODE
                ;
                ; ****************************************
                ;
                ;
                ;
                ;
                ;
1011 60         FIXCHR RTS              SINCE OSI DISPLAY CODES ARE
                ;                       THE SAME AS THE CORRESPONDING
                ;                       ASCII CHARACTERS, NO CONVERSION
                ;                       IS NECESSARY; FIXCHR IS A DUMMY.
```

Appendix C14:

System Data Block for the PET 2001

```
                  ;       APPENDIX C14: ASSEMBLER LISTING OF
                  ;          SYSTEM DATA BLOCK
                  ;           FOR THE PET 2001
                  ;
                  ;
                  ;
                  ;
                  ;     SEE APPENDIX B2 OF BEYOND GAMES: SYSTEM
                  ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                  ;
                  ;
                  ;                  BY KEN SKIER
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ;
                  ; ********************************************
                  ;
                  ;         SCREEN PARAMETERS
                  ;
                  ; ********************************************
                  ;
                  ;
                  ;
                  ;
                  ;
1000                    *=$1000
                  ;
                  ;
                  ;
                  ;
                  ;
1000 0080         HOME   .WORD $8000    THIS IS THE ADDRESS OF THE
                  ;                     CHARACTER IN THE UPPER LEFT
                  ;                     CORNER OF THE SCREEN.
1002 28           ROWINC .BYTE $28      ADDRESS DIFFERENCE FROM ONE
                  ;                     ROW TO THE NEXT.
1003 27           TVCOLS .BYTE 39       NUMBER OF COLUMNS ON SCREEN,
                  ;                     COUNTING FROM ZERO.
1004 18           TVROWS .BYTE 24       NUMBER OF ROWS ON SCREEN,
                  ;                     COUNTING FROM ZERO.
1005 83           HIPAGE .BYTE $83      HIGHEST PAGE IN SCREEN MEMORY.
1006 20           BLANK  .BYTE $20      PET DISPLAY CODE FOR A BLANK.
                  ;                     (IN NORMAL VIDEO MODE.)
1007 1E           ARROW  .BYTE $1E      PET DISPLAY CODE FOR UP-ARROW.
                  ;
                  ;
                  ;
                  ;
```

```
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;             INPUT/OUTPUT VECTORS
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;
                ;
1008 2A10     ROMKEY .WORD PETKEY   POINTER TO ROUTINE THAT GETS
                ;                   AN ASCII CHARACTER FROM THE
                ;                   KEYBOARD.  (NOTE: PETKEY
                ;                   CALLS A ROM SUBROUTINE, BUT
                ;                   PETKEY IS NOT A PET ROM
                ;                   SUBROUTINE.)
                ;
                ;
100A D2FF     ROMTVT .WORD $FFD2    POINTER TO ROUTINE TO PRINT
                ;                   AN ASCII CHARACTER ON THE SCREEN
                ;
                ;
100C 1010     ROMPRT .WORD DUMMY    POINTER TO ROUTINE TO SEND AN
                ;                   ASCII CHARACTER TO THE PRINTER
                ;                   (SET TO DUMMY UNTIL YOU MAKE
                ;                   IT POINT TO THE CHARACTER-
                ;                   OUTPUT ROUTINE THAT DRIVES
                ;                   YOUR PRINTER.)
                ;
                ;
100E 1010     USROUT .WORD DUMMY    POINTER TO USER-WRITTEN OUTPUT
                ;                   ROUTINE.  (SET HERE TO DUMMY
                ;                   UNTIL YOU SET IT TO POINT
                ;                   TO YOUR OWN CHARACTER-OUTPUT
                ;                   ROUTINE.)
                ;
                ;
1010 60       DUMMY  RTS            THIS IS A DUMMY SUBROUTINE.
                ;                   IT DOES NOTHING BUT RETURN.
                ;
                ;
                ;
                ;
                ;
                ;
                ; ********************************************
                ;
                ;      CONVERT ASCII CHARACTER TO DISPLAY CODE
                ;
                ; ********************************************
                ;
                ;
```

```
1190                ;
1200                ;
1210                ;
1220 1011 297F   FIXCHR AND #$7F        CLEAR BIT 7, TO MAKE IT
1230                ;                   A LEGAL ASCII CHARACTER.
1240 1013 38            SEC             PREPARE TO COMPARE.
1250 1014 C940          CMP #$40        IS IT LESS THAN $40? (IS
1260                ;                   IT A NUMBER OR PUNCTUATION
1270                ;                   MARK?)
1280 1016 9011          BCC FIXEND      IF SO, NO CONVERSION NEEDED.
1290                ;
1300 1018 C960          CMP #$60        IS IT BETWEEN $40 AND $60?
1310                ;
1320 101A 900A          BCC SUB.40      IF SO, SUBTRACT $40 TO
1330                ;                   CONVERT FROM ASCII TO PET.
1340                ;
1350                ;                   IT'S >= $60, SO WE MUST
1370 101C A20E          LDX #14         SET PET DISPLAY MODE FOR
1380 101E 8D4CE8        STA 59468       CHARACTER SET THAT INCLUDES
1390                ;                   LOWER CASE ALPHA CHARACTERS.
1400 1021 E920          SBC #$20        SUBTRACT $20 TO CONVERT
1410                ;                   LOWER CASE ASCII TO PET CODE.
1420 1023 18            CLC
1430 1024 9003          BCC FIXEND
1435                ;
1440 1026 38     SUB.40 SEC             PREPARE TO SUBTRACT.
1450 1027 E940          SBC #$40        SUBTRACT $40 TO CONVERT ASCII
1460                ;                   UPPER CASE CHAR TO PET CODE.
1470 1029 60     FIXEND RTS             RETURN, WITH A HOLDING
1480                ;                   PET DISPLAY CODE FOR ASCII
1490                ;                   ORIGINALLY IN A.
1500                ;
1510                ;
1520                ;
1530                ;
1540                ;
1550                ; ********************************************
1560                ;
1570                ;    GET AN ASCII CHARACTER FROM THE KEYBOARD
1580                ;
1590                ; ********************************************
1600                ;
1610                ;
1620                ;
1630                ;
1640 102A 20E4FF PETKEY JSR $FFE4       SCAN THE PET KEYBOARD
1650 102D 297F          AND #$7F        CLEAR BIT 7, TO BE SURE
1660                ;                   IT'S A LEGAL ASCII CHARACTER.
1670 102F F0F9          BEQ PETKEY      ZERO MEANS NO KEY, SO
1680                ;                   SCAN AGAIN.
1690                ;
1700 1031 60            RTS             RETURN WITH ASCII CHARACTER
1710                ;                   FROM THE KEYBOARD.
```

Appendix C15:

System Data Block for the Apple II

```
                ;        APPENDIX C15: ASSEMBLER LISTING OF
                ;           SYSTEM DATA BLOCK
                ;              FOR THE APPLE II
                ;
                ;
                ;
                ;
                ;      SEE APPENDIX B3 OF BEYOND GAMES: SYSTEM
                ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
                ;
                ;
                ;                  BY KEN SKIER
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ;
                ; *********************************************
                ;
                ;          SCREEN PARAMETERS
                ;
                ; *********************************************
                ;
                ;
                ;
                ;
                ;
1000                    *=$1000
                ;
                ;
                ;
                ;
                ;
1000 0004      HOME    .WORD $0400    THIS IS THE ADDRESS OF THE
                ;                     CHARACTER IN THE UPPER LEFT
                ;                     CORNER OF THE SCREEN.
                ;                     (WHEN YOU ARE DISPLAYING
                ;                     LOW-RESOLUTION GRAPHICS AND
                ;                     TEXT PAGE 1.)
1002 80        ROWINC .BYTE $80       ADDRESS DIFFERENCE FROM ONE
                ;                     ROW TO THE NEXT.
1003 27        TVCOLS .BYTE 39        NUMBER OF COLUMNS ON SCREEN,
                ;                     COUNTING FROM ZERO.
1004 07        TVROWS .BYTE 7         NUMBER OF ROWS ON SCREEN,
                ;                     COUNTING FROM ZERO.
1005 07        HIPAGE .BYTE $07       HIGHEST PAGE IN SCREEN MEMORY.
                ;                     (WITH LOW-RES PAGE 1 SELECTED.)
1006 A0        BLANK  .BYTE $A0       APPLE II DISPLAY CODE FOR
                ;                     A BLANK: A DARK BOX, USED AS
                ;                     A SPACE WHEN APPLE II IS IN
```

```
                   ;                       NORMAL DISPLAY MODE (WHITE
                   ;                       CHARACTERS ON A DARK
                   ;                       BACKGROUND.)
1007 DE            ARROW  .BYTE $DE        APPLE II DISPLAY CODE FOR
                   ;                       A CARAT (USED BECAUSE APPLE
                   ;                       II HAS NO UP-ARROW.)
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; **********************************************
                   ;
                   ;              INPUT/OUTPUT VECTORS
                   ;
                   ; **********************************************
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
1008 1410          ROMKEY .WORD APLKEY     POINTER TO ROUTINE THAT GETS
                   ;                       AN ASCII CHARACTER FROM THE
                   ;                       KEYBOARD.  (NOTE: APLKEY
                   ;                       CALLS A ROM SUBROUTINE, BUT
                   ;                       APLKEY IS NOT AN APPLE ROM
                   ;                       SUBROUTINE.)
                   ;
                   ;
100A 1A10          ROMTVT .WORD APLTVT     POINTER TO ROUTINE TO PRINT
                   ;                       AN ASCII CHARACTER ON THE SCREEN
                   ;
                   ;
100C 1010          ROMPRT .WORD DUMMY      POINTER TO ROUTINE TO SEND AN
                   ;                       ASCII CHARACTER TO THE PRINTER
                   ;                       (SET TO DUMMY UNTIL YOU MAKE
                   ;                       IT POINT TO THE CHARACTER-
                   ;                       OUTPUT ROUTINE THAT DRIVES
                   ;                       YOUR PRINTER.)
                   ;                            YOU MAY WISH TO
                   ;                       SET ROMPRT SO IT POINTS TO
                   ;                       $FDED, THE APPLE II'S
                   ;                       GENERAL CHARACTER OUTPUT
                   ;                       ROUTINE.  $FDED WILL PRINT TO
                   ;                       A PRINTER IF YOU TELL
                   ;                       YOUR APPLE II ROM SOFTWARE
                   ;                       TO SELECT  YOUR PRINTER AS
                   ;                       AN OUTPUT DEVICE.  DO THAT
                   ;                       IN BASIC BY TYPING "PR #N",
                   ;                       WHERE N IS THE NUMBER OF THE
                   ;                       SLOT HOLDING THE CIRCUIT CARD
                   ;                       THAT DRIVES YOUR PRINTER.
```

```
                   ;
                   ;
                   ;
     100E 1010  USROUT .WORD DUMMY    POINTER TO USER-WRITTEN OUTPUT
                   ;                  ROUTINE.  (SET HERE TO DUMMY
                   ;                  UNTIL YOU SET IT TO POINT
                   ;                  TO YOUR OWN CHARACTER-OUTPUT
                   ;                  ROUTINE.)
                   ;
                   ;
     1010 60    DUMMY  RTS            THIS IS A DUMMY SUBROUTINE.
                   ;                  IT DOES NOTHING BUT RETURN.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ********************************************
                   ;
                   ;      CONVERT ASCII CHARACTER TO DISPLAY CODE
                   ;
                   ; ********************************************
                   ;
                   ;
                   ;
                   ;
                   ;
     1011 0980  FIXCHR ORA #$80       SET BIT 7, SO CHARACTER
                   ;                  WILL DISPLAY IN NORMAL MODE.
     1013 60           RTS            RETURN.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ********************************************
                   ;
                   ;    GET AN ASCII CHARACTER FROM THE KEYBOARD
                   ;
                   ; ********************************************
                   ;
                   ;
                   ;
                   ;
     1014 2035FD APLKEY JSR $FD35     GET KEYBOARD CHARACTER WITH
                   ;                  BIT 7 SET.
     1017 297F         AND #$7F       CLEAR BIT 7.
                   ;
     1019 60           RTS            RETURN WITH ASCII CHARACTER
                   ;                  FROM THE KEYBOARD.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
```

```
                   ;
                   ; ********************************************
                   ;
                   ;     PRINT AN ASCII CHARACTER ON THE SCREEN
                   ;
                   ; ********************************************
                   ;
                   ;
                   ;
                   ;
                   ;
101A 0980         APLTVT ORA #$80        SET BIT 7 SO CHARACTER WILL
                   ;                     PRINT IN NORMAL MODE.
101C 20FDFB              JSR $FBFD       CALL APPLE II ROM ROUTINE TO
                   ;                     PRINT A CHARACTER TO SCREEN.
101F 60                  RTS             RETURN TO CALLER.
```

Appendix C16:

System Data Block for the Atari 800

```
10                  ;       APPENDIX C16: ASSEMBLER LISTING OF
20                  ;          SYSTEM DATA BLOCK
30                  ;             FOR THE ATARI 800
40                  ;
50                  ;
60                  ;
70                  ;
80                  ;     SEE APPENDIX B4 OF BEYOND GAMES: SYSTEM
90                  ; SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER
100                 ;
110                 ;
120                 ;                 BY KEN SKIER
130                 ;
140                 ;
150                 ;
160                 ;
170                 ;
180                 ;
190                 ;
200                 ;
210                 ;
220                 ;
230                 ;
240                 ;
250                 ;
260                 ; ********************************************
270                 ;
280                 ;        EXTERNAL ADDRESSES
290                 ;
300                 ; ********************************************
310                 ;
320                 ;
330                 ;
340                 ;
350                 ;
360                 ;
370 0000=                   TV.PTR=0
380                 ;
390 1100=                   TVSUBS=$1100
400 1113=                   CLR.XY=TVSUBS+$13
410 112B=                   TVHOME=TVSUBS+$2B
420 113C=                   TVTOXY=TVSUBS+$3C
430 1176=                   TVDOWN=TVSUBS+$76
440 11C4=                   TVPUSH=TVSUBS+$C4
450 11D3=                   TV.POP=TVSUBS+$D3
460 117C=                   VUCHAR=TVSUBS+$7C
470                 ;
480 1500=                   HEX.PG=$1500
490 1552=                   SA=HEX.PG+$52
500 1554=                   EA=SA+2
510                 ;
520 1700=                   MOV.PG=$1700
530 17B2=                   DEST=MOV.PG+$B2
540 17D6=                   MOV.EA=MOV.PG+$D6
550                 ;
560                 ;
570                 ;
```

```
                ;
                ;
                ;
                ; ********************************************
                ;
                ;         SCREEN PARAMETERS
                ;
                ; ********************************************
                ;
                ;
                ;
                ;
                ;
1000                    *=$1000
                ;
                ;
                ;
                ;
1000 427C       HOME    .WORD $7C42    ADDRESS OF THE
                ;                      CHARACTER IN THE UPPER LEFT
                ;                      CORNER OF THE SCREEN.
                ;                      (FOR AN ATARI 800 W/32K RAM,
                ;                      IN SCREEN MODE 0.)
                ;                      YOU MUST USE SCREEN MODE 0.
                ;                      APPENDIX B4 INCLUDES A BASIC
                ;                      PROGRAM TO START THE VISIBLE
                ;                      MONITOR.  IT SETS HOME FOR
                ;                      YOUR SYSTEM.
                ;         NOTE:        IF HOME IS LESS THAN $2000
                ;                      (8192 DECIMAL), THE SCREEN
                ;                      WILL INTERFERE WITH THE
                ;                      SOFTWARE IN THIS BOOK.
                ;
                ;                      IF YOU TRY TO RUN THIS
                ;                      SOFTWARE ON AN 8K SYSTEM, DON'T
                ;                      USE THE DISASSEMBLER OR THE
                ;                      SIMPLE TEXT EDITOR, BECAUSE
                ;                      SCREEN OPERATIONS WILL WRITE
                ;                      OVER THEM, AND THEY'LL CRASH.
                ;
1002 28         ROWINC .BYTE 40        ADDRESS DIFFERENCE FROM ONE
                ;                      ROW TO THE NEXT.
1003 27         TVCOLS .BYTE 39        NUMBER OF COLUMNS ON SCREEN,
                ;                      COUNTING FROM ZERO.
1004 17         TVROWS .BYTE 23        NUMBER OF ROWS ON SCREEN,
                ;                      COUNTING FROM ZERO.
1005 7F         HIPAGE .BYTE $7F       HIGHEST PAGE IN SCREEN
                ;                      MEMORY.  LIKE HOME, HIPAGE
                ;                      VARIES ACCORDING TO THE
                ;                      AMOUNT OF RAM IN YOUR ATARI.
                ;                      HIPAGE IS SET FOR YOUR SYSTEM
                ;                      WHEN YOU RUN THE BASIC PROGRAM
                ;                      IN APPENDIX B4 TO START
                ;                      THE VISIBLE MONITOR.
                ;
1006 00         BLANK   .BYTE 0        ATARI DISPLAY CODE FOR A BLANK
1007 7B         ARROW   .BYTE $7B      ATARI DISPLAY CODE FOR
                                       AN UP-ARROW.
                ;
```

```
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; *********************************************
                   ;
                   ;            INPUT/OUTPUT VECTORS
                   ;
                   ; *********************************************
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
1008 2810   ROMKEY .WORD ATRKEY   POINTER TO ROUTINE THAT GETS
                   ;              AN ASCII CHARACTER FROM THE
                   ;              KEYBOARD.
                   ;
                   ;
100A 3610   ROMTVT .WORD TVTSIN   POINTER TO ROUTINE TO PRINT
                   ;              AN ASCII CHARACTER ON THE SCREEN
                   ;
                   ;
100C 1010   ROMPRT .WORD DUMMY    POINTER TO ROUTINE TO SEND AN
                   ;              ASCII CHARACTER TO THE PRINTER
                   ;              (SET TO DUMMY UNTIL YOU MAKE
                   ;              IT POINT TO THE CHARACTER-
                   ;              OUTPUT ROUTINE
                   ;              THAT DRIVES YOUR PRINTER.
                   ;
                   ;
                   ;
100E 1010   USROUT .WORD DUMMY    POINTER TO USER-WRITTEN OUTPUT
                   ;              ROUTINE.  (SET HERE TO DUMMY
                   ;              UNTIL YOU SET IT TO POINT
                   ;              TO YOUR OWN CHARACTER-OUTPUT
                   ;              ROUTINE.)
                   ;
                   ;
1010 60     DUMMY  RTS            THIS IS A DUMMY SUBROUTINE.
                   ;              IT DOES NOTHING BUT RETURN.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; *********************************************
                   ;
                   ;      CONVERT ASCII CHARACTER TO DISPLAY CODE
                   ;
                   ; *********************************************
                   ;
```

```
                   ;
                   ;
                   ;
                   ;
1011 297F   FIXCHR AND #$7F       CLEAR BIT 7 SO CHARACTER IS
                   ;              A LEGITIMATE ASCII CHARACTER.
1013 38            SEC            PREPARE TO COMPARE.
1014 C920          CMP #$20       IS CHARACTER < $20?
1016 9008          BCC BADCHR     IF SO, IT'S NOT A VIEWABLE
                   ;              ASCII CHARACTER, SO RETURN
                   ;              A BLANK.
                   ;
1018 C960          CMP #$60       IS CHARACTER < $60?
101A 9008          BCC SUB.20     IF SO, SUBTRACT $20 AND RETURN.
101C C97B          CMP #$7B       CHARACTER < $7B?
101E 9007          BCC FIXEND     IF SO, NO CONVERSION IS NEEDED.
                   ;
1020 AD0610 BADCHR LDA BLANK      THE CHARACTER IS NOT A
                   ;              VIEWABLE ASCII CHARACTER,
1023 60            RTS            SO RETURN A BLANK.
1024 38     SUB.20 SEC            PREPARE TO SUBTRACT.
1025 E920          SBC #$20       SUBTRACT $20 TO CONVERT ASCII
                   ;              TO ATARI DISPLAY CODE.
1027 60     FIXEND RTS            RETURN WITH ATARI DISPLAY
                   ;              CODE FOR ORIGINAL ASCII
                   ;              CHARACTER.
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ; ***********************************************
                   ;
                   ;    GET AN ASCII CHARACTER FROM THE KEYBOARD
                   ;
                   ; ***********************************************
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
                   ;
1028 ADFC02 ATRKEY LDA $02FC      HAS A KEY BEEN DEPRESSED?
102B C9FF          CMP #$FF       $FF MEANS NO KEY.
102D F0F9          BEQ ATRKEY     IF NOT, LOOK AGAIN.
                   ;
                   ;              A KEY HAS GONE DOWN.
                   ;              ACCUMULATOR HOLDS ITS
                   ;              HARDWARE KEY-CODE.
102F A8            TAY            PREPARE TO USE THAT CODE AS
                   ;              AS AN INDEX.
                   ;
                   ;
```

```
2320 1030 B9000F        LDA ATRKYS,Y   LOOK UP CHARACTER FOR THAT
2330                ;,                 KEY AND SHIFT STATE.
2340 1033 60            RTS            RETURN WITH ASCII CHARACTER
2350                ;                  FOR THAT KEY AND SHIFT STATE.
2360                ;
2370                ;
2380                ;
2390                ;
2400                ;
2410                ;
2420                ;
2430                ; ********************************************
2440                ;
2450                ;       PRINT AN ASCII CHARACTER ON THE SCREEN
2460                ;
2470                ; ********************************************
2480                ;
2490                ;
2500                ;
2510 000D=              CR=$0D         ASCII CARRIAGE RETURN.
2520 000A=              LF=$0A         ASCII LINEFEED CHARACTER.
2530                ;
2540                ;
2550                ;
2560 1034 00       TVCHAR .BYTE 0      THIS BYTE HOLDS CHARACTER
2570                ;                  TO BE DISPLAYED. (ALSO,
2580                ;                  CHARACTER MOST RECENTLY
2590                ;                  DISPLAYED, USING TVTSIM.)
2600 1035 00       TV.COL .BYTE 0      THIS BYTE HOLDS COLUMN IN
2610                ;                  WHICH CHARACTER WILL NEXT
2620                ;                  APPEAR.  WE MAY THINK OF IT
2630                ;                  AS THE POSITION OF AN
2640                ;                  ELECTRONIC "PRINT-HEAD".
2650                ;
2660                ;
2670                ;
2680 1036 C90D     TVTSIM CMP #CR      IS CHARACTER AN ASCII
2690                ;                  CARRIAGE RETURN?
2700 1038 D006          BNE LFTEST     IF NOT, PERFORM NEXT TEST.
2710 103A A900     RESET  LDA #0       RESET TV COLUMN TO
2720 103C 8D3510        STA TV.COL     LEFT MARGIN AND
2730 103F 60            RTS            RETURN.
2740                ;
2750 1040 C90A     LFTEST CMP #LF      IS IT A LINEFEED CHARACTER?
2760 1042 D003          BNE CHSAVE  IF NOT, HANDLE IT AS A CHARACTER
2770 1044 4C800E        JMP SCROLL     SCROLL TEXT UP FOR A LINEFEED.
2780                ;
2790                ;                  SINCE IT'S NOT CR OR LF,
2800 1047 8D3410   CHSAVE STA TVCHAR   LET'S SAVE IT.
2810 104A 20C411        JSR TVPUSH     SAVE ZERO PAGE BYTES WE'LL USE.
2820                ;
2830 104D AC0410        LDY TVROWS     SET TV.PTR TO CURRENT
2840 1050 AE3510        LDX TV.COL     POSITION OF "PRINT-HEAD".
2850 1053 203C11        JSR TVTOXY
2860                ;
2870 1056 AD3410        LDA TVCHAR     GET CHARACTER TO BE DISPLAYED.
2880 1059 207C11        JSR VUCHAR     SHOW IT.
2890 105C EE3510        INC TV.COL     ADVANCE "PRINT-HEAD" TO NEXT
```

```
2900                   ;                          SCREEN POSITION.
2910                   ;
2920 105F AD3510              LDA TV.COL     HAS "PRINT-HEAD" REACHED
2930 1062 CD0310              CMP TVCOLS     RIGHT EDGE OF SCREEN?
2940                   ;
2950 1065 D006                BNE TVTEND     IF NOT, PREPARE TO RETURN.
2960 1067 203A10              JSR RESET      IF SO, RESET "PRINT-HEAD" TO
2970 106A 20800E              JSR SCROLL     LEFT MARGIN AND SCROLL TEXT.
2980 106D 20D311 TVTEND JSR TV.POP     RESTORE ZERO PAGE BYTES
2990                   ;
3000 1070 60                  RTS            WE USED, AND RETURN.
3010                   ;
3020                   ;
3030                   ;
3040                   ;
3050                   ;
3060                   ;
3070                   ; ********************************************
3080                   ;
3090                   ;          SCROLL TEXT UP ON SCREEN
3100                   ;
3110                   ; ********************************************
3120                   ;
3130                   ;
3140                   ;
3150                   ;
3160 0E80                        *=$0E80
3170                   ;
3180                   ;
3190                   ;
3200                   ;
3210                   ;
3220 0E80 20C411 SCROLL JSR TVPUSH     SAVE ZERO PAGE BYTES WE'LL
3230                   ;                     USE.
3240                   ;                     SCROLLING IS SIMPLY MOVING
3250                   ;                     THE CONTENTS OF SCREEN MEMORY
3260                   ;                     UP BY ONE ROW. BEFORE WE
3270                   ;                     MOVE ANYTHING, HOWEVER, LET'S
3280                   ;                     SAVE SA, EA, AND DEST--
3290                   ;                     THE MOVE PARAMETERS.
3300                   ;
3310 0E83 ADB317              LDA DEST+1
3320 0E86 48                  PHA
3330 0E87 ADB217              LDA DEST
3340 0E8A 48                  PHA
3350 0E8B AD5515              LDA EA+1
3360 0E8E 48                  PHA
3370 0E8F AD5415              LDA EA
3380 0E92 48                  PHA
3390 0E93 AD5315              LDA SA+1
3400 0E96 48                  PHA
3410 0E97 AD5215              LDA SA
3420 0E9A 48                  PHA
3430                   ;                     NOW SA, EA, AND DEST ARE SAVED.
3440                   ;
3450 0E9B 202B11              JSR TVHOME     SET TV.PTR TO HOME POSITION.
3460 0E9E A500                LDA TV.PTR     SET DEST=HOME, SINCE WE'LL
3470 0EA0 8DB217              STA DEST       MOVE THE CONTENTS OF SCREEN
```

```
3480 0EA3 A501          LDA TV.PTR+1   MEMORY TOWARDS THE HOME
3490 0EA5 8DB317        STA DEST+1     ADDRESS.
3500               ;
3510 0EA8 207611        JSR TVDOWN     SET SA=ADDRESS OF SCREEN
3520 0EAB A500          LDA TV.PTR     POSITION AT COLUMN 0, ROW 1.
3530 0EAD 8D5215        STA SA         THAT MARKS THE START OF
3540 0EB0 A501          LDA TV.PTR+1   OF THE BLOCK TO BE MOVED.
3550 0EB2 8D5315        STA SA+1
3560               ;
3570 0EB5 AE0310        LDX TVCOLS     SET EA=ADDRESS OF POSITION
3580 0EB8 AC0410        LDY TVROWS     IN BOTTOM RIGHT CORNER OF
3590 0EBB 203C11        JSR TVTOXY     THE SCREEN.
3600 0EBE A500          LDA TV.PTR
3610 0EC0 8D5415        STA EA
3620 0EC3 A501          LDA TV.PTR+1   EA WILL MARK THE END OF
3630 0EC5 8D5515        STA EA+1       THE BLOCK TO BE  MOVED.
3640               ;
3650               ;                   NOW SA, EA, AND DEST SPECIFY
3660               ;                   THE BLOCK TO BE MOVED, AND
3670               ;                   ITS DESTINATION.
3680 0EC8 20D617        JSR MOV.EA     MOVE THE BLOCK.
3690 0ECB AC0410        LDY TVROWS     SET TV.PTR TO BOTTOM LEFT
3700 0ECE A200          LDX #0         CORNER OF SCREEN.
3710 0ED0 203C11        JSR TVTOXY
3720 0ED3 AE0310        LDX TVCOLS     CLEAR THIS ROW.
3730 0ED6 A001          LDY #1
3740 0ED8 201311        JSR CLR.XY
3750 0EDB 68            PLA         RESTORE THE MOVE
3760 0EDC 8D5215        STA SA         PARAMETERS: SA, EA, AND DEST.
3770 0EDF 68            PLA
3780 0EE0 8D5315        STA SA+1
3790 0EE3 68            PLA
3800 0EE4 8D5415        STA EA
3810 0EE7 68            PLA
3820 0EE8 8D5515        STA EA+1
3830 0EEB 68            PLA
3840 0EEC 8DB217        STA DEST
3850 0EEF 68            PLA
3860 0EF0 8DB317        STA DEST+1
3870 0EF3 20D311        JSR TV.POP     RESTORE ZERO PAGE BYTES WE
3880               ;                   USED.
3890 0EF6 60            RTS            RETURN.
3900               ;
3910               ;
3920               ;
3930               ;
3940               ;
3950               ;
3960               ;
3970               ;
3980               ;
3990               ;
4000               ; ********************************************
4010               ;
4020               ;       KEYBOARD DEFINITION TABLE
4030               ;
4040               ; ********************************************
4050               ;
```

```
4060                ;
4070                ;
4080                ;
4090                ;
4100                ;
4110 0F00                  *=$0F00
4120                ;
4130                ;
4140                ;
4150                ;
4160                ;
4170 0027=                 APOSTR=$27      ASCII APOSTROPHE.
4180 005E=                 CARAT=$5E       ASCII CARAT.
4190 001B=                 ESC=$1B         ASCII ESCAPE CHARACTER.
4200 0020=                 SPACE=$20       ASCII SPACE.
4210 0009=                 TAB=9           ASCII TAB CHARACTER.
4220 005B=                 BACKSL=$5B      ASCII BACKSLASH CHARACTER.
4230 0008=                 BACKSP=8        ASCII BACKSPACE CHARACTER.
4240 005A=                 LBRAKT=$5A      ASCII LEFT BRACKET.
4250 005D=                 RBRAKT=$5D      ASCII RIGHT BRACKET.
4260 007F=                 DELETE=$7F      ASCII DELETE CHARACTER.
4270                ;
4280                ;
4290                ;
4300 0F00 6C        ATRKYS .BYTE 'lj;',0,0,'k+*o',0,'pu',CR,'i-='
4300 0F01 6A
4300 0F02 3B
4300 0F03 00
4300 0F04 00
4300 0F05 6B
4300 0F06 2B
4300 0F07 2A
4300 0F08 6F
4300 0F09 00
4300 0F0A 70
4300 0F0B 75
4300 0F0C 0D
4300 0F0D 69
4300 0F0E 2D
4300 0F0F 3D
4310 0F10 76               .BYTE 'v',0,'c',0,0,'bxz4',0,'36',ESC,'521'
4310 0F11 00
4310 0F12 63
4310 0F13 00
4310 0F14 00
4310 0F15 62
4310 0F16 78
4310 0F17 7A
4310 0F18 34
4310 0F19 00
4310 0F1A 33
4310 0F1B 36
4310 0F1C 1B
4310 0F1D 35
4310 0F1E 32
4310 0F1F 31
4320 0F20 2C               .BYTE ', .n',0,'m/',0,'r',0,'ey',TAB,'twq'
4320 0F21 20
```

```
4320 0F22 2E
4320 0F23 6E
4320 0F24 00
4320 0F25 6D
4320 0F26 2F
4320 0F27 00
4320 0F28 72
4320 0F29 00
4320 0F2A 65
4320 0F2B 79
4320 0F2C 09
4320 0F2D 74
4320 0F2E 77
4320 0F2F 71
4330 0F30 39          .BYTE '9',0,'07',BACKSP,'8<>fhd',0,0,'gsa'
4330 0F31 00
4330 0F32 30
4330 0F33 37
4330 0F34 08
4330 0F35 38
4330 0F36 3C
4330 0F37 3E
4330 0F38 66
4330 0F39 68
4330 0F3A 64
4330 0F3B 00
4330 0F3C 00
4330 0F3D 67
4330 0F3E 73
4330 0F3F 61
4340               ;
4350               ;
4360               ;              FOLLOWING 64 BYTES CONTAIN
4370               ;              ASCII CODES FOR SHIFTED KEYS.
4380               ;
4390               ;
4400 0F40 4C          .BYTE 'LJ:',0,0,'K',BACKSL,CARAT
4400 0F41 4A
4400 0F42 3A
4400 0F43 00
4400 0F44 00
4400 0F45 4B
4400 0F46 5B
4400 0F47 5E
4410 0F48 4F          .BYTE 'O',0,'PU',CR,'I-='
4410 0F49 00
4410 0F4A 50
4410 0F4B 55
4410 0F4C 0D
4410 0F4D 49
4410 0F4E 2D
4410 0F4F 3D
4420 0F50 56          .BYTE 'V',0,'C',0,0,'BXZ4',0,'36',ESC.'%"!'
4420 0F51 00
4420 0F52 43
4420 0F53 00
4420 0F54 00
4420 0F55 42
```

```
4420 0F56 58
4420 0F57 5A
4420 0F58 34
4420 0F59 00
4420 0F5A 33
4420 0F5B 36
4420 0F5C 1B
4420 0F5D 25
4420 0F5E 22
4420 0F5F 21
4430 0F60 5A                 .BYTE LBRAKT,SPACE,RBRAKT,'N',0,'M?',0
4430 0F61 20
4430 0F62 5D
4430 0F63 4E
4430 0F64 00
4430 0F65 4D
4430 0F66 3F
4430 0F67 00
4440 0F68 52                 .BYTE 'R',0,'EY',TAB,'TWQ'
4440 0F69 00
4440 0F6A 45
4440 0F6B 59
4440 0F6C 09
4440 0F6D 54
4440 0F6E 57
4440 0F6F 51
4450 0F70 28                 .BYTE '(',0,')',APOSTR,DELETE,'@',0,0
4450 0F71 00
4450 0F72 29
4450 0F73 27
4450 0F74 7F
4450 0F75 40
4450 0F76 00
4450 0F77 00
4460 0F78 46                 .BYTE 'FHD',0,0,'GSA'
4460 0F79 48
4460 0F7A 44
4460 0F7B 00
4460 0F7C 00
4460 0F7D 47
4460 0F7E 53
4460 0F7F 41
4470           ;
4480           ;                        THE FOLLOWING 128 BYTES
4490           ;                        CONTAIN CHARACTER CODES FOR
4500           ;                        CONTROL SHIFTED KEYS.  EDITOR
4510           ;                        FUNCTION KEYS ARE DEFINED.
4520           ;
4530           ;
4540 0F80 00                 .BYTE 0,0,0,0,0,0,0,0,0,0,$10,0,0,0,0,0
4540 0F81 00
4540 0F82 00
4540 0F83 00
4540 0F84 00
4540 0F85 00
4540 0F86 00
4540 0F87 00
4540 0F88 00
```

```
4540 0F89 00
4540 0F8A 10
4540 0F8B 00
4540 0F8C 00
4540 0F8D 00
4540 0F8E 00
4540 0F8F 00
4550 0F90 00                    .BYTE 0,0,3,0,0,0,0,0,0,0,0.0.0,0,0.0
4550 0F91 00
4550 0F92 03
4550 0F93 00
4550 0F94 00
4550 0F95 00
4550 0F96 00
4550 0F97 00
4550 0F98 00
4550 0F99 00
4550 0F9A 00
4550 0F9B 00
4550 0F9C 00
4550 0F9D 00
4550 0F9E 00
4550 0F9F 00
4560 0FA0 00                    .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0.0,0
4560 0FA1 00
4560 0FA2 00
4560 0FA3 00
4560 0FA4 00
4560 0FA5 00
4560 0FA6 00
4560 0FA7 00
4560 0FA8 00
4560 0FA9 00
4560 0FAA 00
4560 0FAB 00
4560 0FAC 00
4560 0FAD 00
4560 0FAE 00
4560 0FAF 00
4570 0FB0 00                    .BYTE 0,0,0,0,0,0,0,0,6,0,0,0,0,0,0,0
4570 0FB1 00
4570 0FB2 00
4570 0FB3 00
4570 0FB4 00
4570 0FB5 00
4570 0FB6 00
4570 0FB7 00
4570 0FB8 06
4570 0FB9 00
4570 0FBA 00
4570 0FBB 00
4570 0FBC 00
4570 0FBD 00
4570 0FBE 00
4570 0FBF 00
4580 0FC0 00                    .BYTE 0,0.0.0,0,0.0.0,0,0,0,0,0,0,0,0
4580 0FC1 00
4580 0FC2 00
```

```
4580 0FC3 00
4580 0FC4 00
4580 0FC5 00
4580 0FC6 00
4580 0FC7 00
4580 0FC8 00
4580 0FC9 00
4580 0FCA 00
4580 0FCB 00
4580 0FCC 00
4580 0FCD 00
4580 0FCE 00
4580 0FCF 00
4590 0FD0 00              .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4590 0FD1 00
4590 0FD2 00
4590 0FD3 00
4590 0FD4 00
4590 0FD5 00
4590 0FD6 00
4590 0FD7 00
4590 0FD8 00
4590 0FD9 00
4590 0FDA 00
4590 0FDB 00
4590 0FDC 00
4590 0FDD 00
4590 0FDE 00
4590 0FDF 00
4600 0FE0 00              .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4600 0FE1 00
4600 0FE2 00
4600 0FE3 00
4600 0FE4 00
4600 0FE5 00
4600 0FE6 00
4600 0FE7 00
4600 0FE8 00
4600 0FE9 00
4600 0FEA 00
4600 0FEB 00
4600 0FEC 00
4600 0FED 00
4600 0FEE 00
4600 0FEF 00
4610 0FF0 00              .BYTE 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4610 0FF1 00
4610 0FF2 00
4610 0FF3 00
4610 0FF4 00
4610 0FF5 00
4610 0FF6 00
4610 0FF7 00
4610 0FF8 00
4610 0FF9 00
4610 0FFA 00
4610 0FFB 00
4610 0FFC 00
```

```
4610 0FFD 00
4610 0FFE 00
4610 0FFF 00
```

Appendix D1:

Screen Utilities

```
APPENDIX D1:            SCREEN UTILITIES

SEE CHAPTER 5 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.


DUMPING $1100-$11FF

        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

1100   20 C4 11 20 2B 11 AE 03 10 AC 04 10 20 13 11 20
1110   D3 11 60 8E 2A 11 98 AA AD 06 10 AC 2A 11 91 00
1120   88 10 FB 20 76 11 CA 10 EF 60 19 A2 00 A0 00 18
1130   90 0A AD 04 10 4A A8 AD 03 10 4A AA 38 EC 03 10
1140   90 03 AE 03 10 38 CC 04 10 90 03 AC 04 10 AD 00
1150   10 85 00 AD 01 10 85 01 08 D8 8A 18 65 00 90 03
1160   E6 01 18 C0 00 F0 0B 18 6D 02 10 90 02 E6 01 88
1170   D0 F5 85 00 28 60 AD 02 10 18 90 05 20 9B 11 A9
1180   01 08 D8 18 65 00 90 02 E6 01 85 00 38 AD 05 10
1190   C5 01 B0 05 AD 01 10 85 01 28 60 20 11 10 A0 00
11A0   91 00 60 48 4A 4A 4A 4A 20 B6 11 20 7C 11 68 20
11B0   B6 11 20 7C 11 60 08 D8 29 0F C9 0A 30 02 69 06
11C0   69 30 28 60 68 AA 68 A8 A5 01 48 A5 00 48 98 48
11D0   8A 48 60 68 AA 68 A8 68 85 00 68 85 01 98 48 8A
11E0   48 60 00 00 00 00 00 00 00 00 00 00 00 00 00 00
11F0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Appendix D2:

Visible Monitor (Top Level and Display Subroutines)

APPENDIX D2: THE VISIBLE MONITOR (TOP LEVEL AND DISPLAY SUBROUTINES)

SEE CHAPTER 6 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTE

DUMPING $1200-$12DF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1200	00	0C	00	00	31	05	12	08	D8	20	12	12	20	E3	12	18
1210	90	F6	20	C4	11	20	25	12	20	34	12	20	5C	12	20	AF
1220	12	20	D3	11	60	A2	02	A0	02	20	3C	11	A2	19	A0	03
1230	20	13	11	60	A2	0D	A0	02	20	3C	11	A0	00	8C	51	12
1240	B9	52	12	20	7C	11	EE	51	12	AC	51	12	C0	0A	D0	F0
1250	60	0A	41	20	20	58	20	20	59	20	20	50	A2	02	A0	03
1260	20	3C	11	AD	05	12	20	A3	11	AD	05	12	20	A3	11	20
1270	7F	11	20	94	12	48	20	A3	11	20	7F	11	68	20	7C	11
1280	20	7F	11	A2	00	BD	01	12	20	A3	11	20	7F	11	E8	E0
1290	04	D0	F2	60	A5	02	48	A6	03	AD	05	12	85	02	AD	06
12A0	12	85	03	A0	00	B1	02	A8	68	85	02	86	03	98	60	A2
12B0	02	A0	04	20	3C	11	AC	00	12	38	C0	07	90	05	A0	00
12C0	8C	00	12	B9	CD	12	A8	AD	07	10	91	00	60	03	06	08
12D0	0B	0E	11	14	00	00	00	00	00	00	00	00	00	00	00	00

Appendix D3:

Visible Monitor (Update Subroutine)

```
APPENDIX D3:        THE VISIBLE MONITOR  (UPDATE SUBROUTINE)

SEE CHAPTER 6 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.


DUMPING $12E0-$13FF

         0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

12E0    6C 08 10 20 E0 12 C9 3E D0 10 EE 00 12 AD 00 12
12F0    C9 07 D0 05 A9 00 8D 00 12 60 C9 3C D0 0B CE 00
1300    12 10 05 A9 06 8D 00 12 60 C9 20 D0 09 EE 05 12
1310    D0 03 EE 06 12 60 C9 0D D0 0C AD 05 12 D0 03 CE
1320    06 12 CE 05 12 60 AE 00 12 E0 02 D0 1B A8 A5 00
1330    48 A6 01 AD 05 12 85 00 AD 06 12 85 01 98 A0 00
1340    91 00 86 01 68 85 00 60 C9 47 D0 23 AC 03 12 AE
1350    02 12 AD 04 12 48 AD 01 12 28 20 6C 13 08 8D 01
1360    12 8E 02 12 8C 03 12 68 8D 04 12 60 6C 05 12 48
1370    20 D5 13 30 4B A8 68 98 AE 00 12 D0 14 A2 03 18
1380    0E 05 12 2E 06 12 CA 10 F6 98 0D 05 12 8D 05 12
1390    60 E0 01 D0 18 29 0F 48 20 94 12 0A 0A 0A 0A 29
13A0    F0 8D AC 13 68 0D AC 13 20 2D 13 60 00 CA CA CA
13B0    A0 03 18 1E 01 12 88 10 F9 1D 01 12 9D 01 12 60
13C0    68 C9 7F D0 04 20 00 11 60 C9 51 D0 04 68 68 28
13D0    60 20 B0 10 60 38 E9 30 90 0F C9 0A 90 0E E9 07
13E0    C9 10 B0 05 38 C9 0A B0 03 A9 FF 60 A2 00 60 00
13F0    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Appendix D4:

Print Utilities

APPENDIX D4: PRINT UTILITIES

SEE CHAPTER 7 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTE

DUMPING $1400-$154F

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1400	FF	FF	00	20	00	00	0C	15	A9	FF	8D	01	14	60	A9	00
1410	8D	01	14	60	A9	FF	8D	00	14	60	A9	00	8D	00	14	60
1420	A9	FF	8D	02	14	60	A9	00	8D	02	14	60	20	08	14	20
1430	14	14	20	20	14	60	20	0E	14	20	1A	14	20	26	14	60
1440	C9	00	F0	24	8D	03	14	AD	01	14	F0	06	AD	03	14	20
1450	69	14	AD	00	14	F0	06	AD	03	14	20	6C	14	AD	02	14
1460	F0	06	AD	03	14	20	6F	14	60	6C	0A	10	6C	0C	10	6C
1470	0E	10	A9	0D	20	40	14	A9	0A	20	40	14	60	A9	20	20
1480	40	14	60	48	4A	4A	4A	4A	20	B6	11	20	40	14	68	20
1490	B6	11	20	40	14	60	A9	20	8E	04	14	48	AE	04	14	F0
14A0	0A	CE	04	14	20	40	14	68	18	90	F0	68	60	8E	04	14
14B0	AE	04	14	F0	09	CE	04	14	20	72	14	18	90	F2	60	8E
14C0	05	14	B5	01	48	B5	00	48	AE	05	14	A1	00	C9	FF	F0
14D0	0C	F6	00	D0	02	F6	01	20	40	14	18	90	EB	68	95	00
14E0	68	95	01	60	68	AA	68	A8	20	12	15	8E	05	12	8C	06
14F0	12	20	0D	13	20	0D	13	20	94	12	C9	FF	F0	06	20	40
1500	14	18	90	F0	AE	05	12	AC	06	12	20	2B	15	98	48	8A
1510	48	60	68	8D	06	14	68	8D	07	14	AD	06	12	48	AD	05
1520	12	48	AD	07	14	48	AD	06	14	48	60	68	8D	06	14	68
1530	8D	07	14	68	8D	05	12	68	8D	06	12	AD	07	14	48	AD
1540	06	14	48	60	00	00	00	00	00	00	00	00	00	00	00	00

Appendix D5:

Two Hexdump Tools

```
APPENDIX D5:            TWO HEXDUMP TOOLS

SEE CHAPTER 8 OF BEYOND .GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER


DUMPING $1550-$17AF

        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

1550   00 04 50 15 AF 17 00 20 08 14 AD 51 15 8D 50 15
1560   AD 05 12 29 F8 8D 05 12 20 72 14 20 72 14 20 A1
1570   15 20 72 14 20 7D 14 20 9A 15 20 0D 13 AD 05 12
1580   29 07 D0 F0 20 72 14 AD 05 12 29 0F D0 03 20 72
1590   14 CE 50 15 D0 D8 20 0E 14 60 20 94 12 20 83 14
15A0   60 AD 06 12 20 83 14 AD 05 12 20 83 14 60 20 C9
15B0   15 20 E9 15 20 A0 17 20 14 14 20 EB 16 20 42 17
15C0   10 FB 20 72 14 20 1A 14 60 20 00 11 20 08 14 20
15D0   E4 14 7F 0D 50 52 49 4E 54 49 4E 47 20 48 45 58
15E0   44 55 4D 50 0D 0A 0A FF 60 20 08 14 20 E4 14 7F
15F0   0D 0A 53 45 54 20 53 54 41 52 54 49 4E 47 20 41
1600   44 44 52 45 53 53 20 41 4E 44 20 50 52 45 53 53
1610   20 22 51 22 2E FF 20 07 12 20 67 16 20 08 14 20
1620   E4 14 7F 0D 0A 53 45 54 20 45 4E 44 20 41 44 44
1630   52 45 53 53 20 41 4E 44 20 50 52 45 53 53 20 22
1640   51 22 2E FF 20 07 12 38 AD 06 12 CD 53 15 90 24
1650   D0 08 AD 05 12 CD 52 15 90 1A AD 06 12 8D 55 15
1660   AD 05 12 8D 54 15 60 AD 06 12 8D 53 15 AD 05 12
1670   8D 52 15 60 20 E4 14 7F 0D 0A 0A 0A 20 45 52 52
1680   4F 52 21 21 21 20 45 4E 44 20 41 44 44 52 45 53
1690   53 20 4C 45 53 53 20 54 48 41 4E 20 53 54 41 52
16A0   54 20 41 44 44 52 45 53 53 2C 20 57 48 49 43 48
16B0   20 49 53 20 FF 20 BB 16 4C 1C 16 A9 24 20 40 14
16C0   AD 53 15 20 83 14 AD 52 15 20 83 14 60 A9 24 20
16D0   40 14 AD 55 15 20 83 14 AD 54 15 20 83 14 60 20
16E0   BB 16 A9 2D 20 40 14 20 CD 16 60 20 E4 14 7F 0D
16F0   0A 0A 44 55 4D 50 49 4E 47 20 FF 20 DF 16 20 72
1700   14 20 E4 14 7F 0A 0A 20 20 20 20 20 20 20 20 30
1710   20 20 31 20 20 32 20 20 33 20 20 34 20 20 35 20
```

```
1720    20 36 20 20 37 20 20 38 20 20 39 20 20 41 20 20
1730    42 20 20 43 20 20 44 20 20 45 20 20 46 0D 0A 0A
1740    FF 60 20 72 14 AD 05 12 48 29 0F 8D 56 15 68 29
1750    F0 8D 05 12 20 A1 15 A2 03 20 96 14 AD 56 15 F0
1760    0D A2 03 20 96 14 20 0D 13 CE 56 15 D0 F3 20 9A
1770    15 20 7D 14 20 83 17 30 09 AD 05 12 29 0F C9 00
1780    D0 EC 60 38 AD 06 12 CD 55 15 90 0B D0 0F 38 AD
1790    05 12 CD 54 15 B0 06 20 0D 13 A9 00 60 A9 FF 60
17A0    AD 52 15 8D 05 12 AD 53 15 8D 06 12 60 00 00 00
```

Appendix D6:

Table-Driven Disassembler (Top Level and Utility Subroutines)

APPENDIX D6: TABLE-DRIVEN DISASSEMBLER (TOP LEVEL AND UTILITY SUBROUTINES)

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $1900-$1A3F

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1900	05	00	00	5A	40	1A	FF	04	10	20	08	14	AD	00	19	8D
1910	01	19	A9	FF	8D	54	15	8D	55	15	20	72	14	20	7D	19
1920	CE	01	19	D0	F8	60	20	1A	14	20	08	14	20	E4	14	7F
1930	0D	0A	20	20	20	20	20	50	52	49	4E	54	49	4E	47	20
1940	44	49	53	41	53	53	45	4D	42	4C	45	52	2E	0D	0A	FF
1950	20	E9	15	20	14	14	20	E4	14	7F	0D	0A	44	49	53	41
1960	53	53	45	4D	42	4C	49	4E	47	20	FF	20	DF	16	20	A0
1970	17	20	72	14	20	7D	19	10	FB	20	1A	14	60	20	94	12
1980	48	20	92	19	20	7D	14	68	20	AF	19	20	01	1A	20	83
1990	17	60	A2	03	8E	02	19	AA	BD	00	1C	AA	BD	50	1B	8E
19A0	03	19	20	40	14	AE	03	19	E8	CE	02	19	D0	EE	60	AA
19B0	BD	00	1D	AA	20	B8	19	60	BD	1B	1B	8D	04	19	E8	BD
19C0	1B	1B	8D	05	19	6C	04	19	20	0D	13	20	9A	15	60	20
19D0	0D	13	20	94	12	48	20	0D	13	20	9A	15	68	20	83	14
19E0	60	A9	28	D0	02	A9	29	20	40	14	60	A9	2C	20	40	14
19F0	A5	58	20	40	14	60	A9	2C	20	40	14	A9	59	20	40	14
1A00	60	8D	07	19	8E	06	19	CA	30	06	20	1A	13	CA	10	FA
1A10	08	D8	38	AD	08	19	E9	04	ED	07	19	28	AA	20	96	14
1A20	20	A1	15	20	7D	14	20	9A	15	20	0D	13	CE	06	19	10
1A30	F2	20	1A	13	20	72	14	60	00	00	00	00	00	00	00	00

Appendix D7:

Table-Driven Disassembler (Addressing Mode Subroutines)

APPENDIX D7: TABLE-DRIVEN DISASSEMBLER (ADDRESSING MODE SUBROUTINES)

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTE

DUMPING $1A40-$1B4F

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1A40	20	CF	19	A2	02	A9	04	60	20	40	1A	20	EB	19	A2	02
1A50	A9	06	60	20	40	1A	20	F6	19	A2	02	A9	06	60	A9	41
1A60	20	40	14	A2	00	A9	01	60	A2	00	A9	00	60	A9	23	20
1A70	40	14	A9	24	20	40	14	20	C8	19	A2	01	A9	04	60	20
1A80	E1	19	20	40	1A	20	E5	19	A9	06	A2	02	60	20	E1	19
1A90	20	E8	1A	20	E5	19	A2	01	A9	08	60	20	E1	19	20	DB
1AA0	1A	20	E5	19	20	F6	19	A2	01	A9	08	60	20	0D	13	20
1AB0	12	15	20	94	12	48	20	0D	13	68	C9	00	10	03	CE	06
1AC0	12	08	D8	18	6D	05	12	90	03	EE	06	12	8D	05	12	28
1AD0	20	A1	15	20	2B	15	A2	01	A9	04	60	A9	00	20	83	14
1AE0	20	C8	19	A2	01	A9	04	60	20	DB	1A	20	EB	19	A2	01
1AF0	A9	06	60	20	DB	1A	20	F6	19	A2	01	A9	06	60	68	68
1B00	68	68	20	83	17	30	0D	20	94	12	C9	FF	F0	06	20	40
1B10	14	18	90	EE	20	72	14	20	83	17	60	68	1A	5E	1A	6D
1B20	1A	DB	1A	E8	1A	F3	1A	40	1A	48	1A	53	1A	68	1A	AC
1B30	1A	8D	1A	9B	1A	7F	1A	FE	1A	00	00	00	00	00	00	00
1B40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Appendix D8:

Table-Driven Disassembler (Tables)

APPENDIX D8: TABLE-DRIVEN DISASSEMBLER (TABLES)

SEE CHAPTER 9 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $1B50-$1DFF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1B50	7F	42	41	44	41	44	43	41	4E	44	41	53	4C	42	43	43
1B60	42	43	53	42	45	51	42	49	54	42	4D	49	42	4E	45	42
1B70	50	4C	42	52	4B	42	56	43	42	56	53	43	4C	43	43	4C
1B80	44	43	4C	49	43	4C	56	43	4D	50	43	50	58	43	50	59
1B90	44	45	43	44	45	58	44	45	59	45	4F	52	49	4E	43	49
1BA0	4E	58	49	4E	59	4A	4D	50	4A	53	52	4C	44	41	4C	44
1BB0	58	4C	44	59	4C	53	52	4E	4F	50	4F	52	41	50	48	41
1BC0	50	48	50	50	4C	41	50	4C	50	52	4F	4C	52	4F	52	52
1BD0	54	49	52	54	53	53	42	43	53	45	43	53	45	44	53	45
1BE0	49	53	54	41	53	54	58	53	54	59	54	41	58	54	41	59
1BF0	54	53	58	54	58	41	54	58	53	54	59	41	54	45	58	FF
1C00	22	6A	01	01	01	6A	0A	01	70	6A	0A	01	01	6A	0A	01
1C10	1F	6A	01	01	01	6A	0A	01	2B	6A	01	01	01	6A	0A	01
1C20	58	07	01	01	16	07	79	01	76	07	79	01	16	07	79	01
1C30	19	07	01	01	01	07	79	01	68	07	01	01	01	07	79	01
1C40	7F	49	01	01	01	49	64	01	6D	49	64	01	55	49	64	01
1C50	25	49	01	01	01	49	64	01	31	49	01	01	01	49	64	01
1C60	82	04	01	01	01	04	7C	01	73	04	7C	01	55	04	7C	01
1C70	28	04	01	01	01	04	7C	01	8E	04	01	01	01	04	7C	AC
1C80	01	91	01	01	97	91	94	01	46	01	A3	01	97	91	94	01
1C90	0D	91	01	01	97	91	94	01	A9	91	A3	01	01	91	01	01
1CA0	61	5B	5E	01	61	5B	5E	01	9D	5B	9A	01	61	5B	5E	01
1CB0	10	5B	01	01	61	5B	5E	01	34	5B	9E	01	61	5B	5E	01
1CC0	3D	37	01	01	3D	37	40	01	52	37	43	01	3D	37	40	01
1CD0	1C	37	01	01	01	37	40	01	2E	37	01	01	01	37	40	01
1CE0	3A	85	01	01	3A	85	4C	01	4F	85	67	01	3A	85	4C	01
1CF0	13	85	01	01	01	85	4C	01	8B	85	01	01	01	85	4C	01
1D00	12	16	00	00	00	06	06	00	12	04	02	00	00	0C	0C	00
1D10	14	18	00	00	00	0E	0E	00	12	10	00	00	00	16	16	00

```
1D20    0C 16 00 00 06 06 06 00 12 04 02 00 0C 0C 0C 00
1D30    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
1D40    12 16 00 00 00 05 06 00 12 0C 02 00 0C 0C 0C 00
1D50    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
1D60    12 16 00 00 00 06 06 00 12 04 02 00 1A 0C 0C 00
1D70    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 1C
1D80    00 16 00 00 06 06 06 00 12 00 12 00 0C 0C 0C 00
1D90    14 18 00 00 08 08 0A 00 12 10 12 00 00 0E 00 00
1DA0    04 16 04 00 06 06 06 00 12 04 12 00 0C 0C 0C 00
1DB0    14 18 00 00 08 08 0A 00 14 10 12 00 0E 0E 10 00
1DC0    04 16 00 00 06 06 06 00 12 04 12 00 0C 0C 0C 00
1DD0    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
1DE0    04 16 00 00 06 06 06 00 12 04 12 00 0C 0C 0C 00
1DF0    14 18 00 00 00 08 08 00 12 10 00 00 00 0E 0E 00
```

Appendix D9:

Move Utilities

APPENDIX D9: MOVE UTILITIES

SEE CHAPTER 10 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER

DUMPING $17B0-$18FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
17B0	C7	00	39	04	20	08	14	20	E4	14	7F	0D	0A	20	20	20
17C0	20	20	4D	4F	56	45	20	54	4F	4F	4C	2E	0D	0A	0A	FF
17D0	20	E9	15	20	B9	18	AE	55	15	38	AD	54	15	ED	52	15
17E0	8D	B0	17	B0	02	CA	38	8A	ED	53	15	8D	B1	17	B0	03
17F0	A9	00	60	A0	03	B9	00	00	48	88	10	F9	38	AD	53	15
1800	CD	B3	17	90	40	D0	18	AD	52	15	CD	B2	17	90	36	D0
1810	0E	A0	00	68	99	00	00	C8	C0	04	D0	F7	A9	FF	60	20
1820	A4	18	A0	00	AE	B1	17	F0	0E	B1	00	91	02	C8	D0	F9
1830	E6	01	E6	03	CA	D0	F2	88	C8	B1	00	91	02	CC	B0	17
1840	D0	F6	4C	11	18	AD	B1	17	F0	48	AC	B1	17	AD	B0	17
1850	38	E9	FF	B0	01	88	AA	84	03	8A	18	6D	52	15	85	00
1860	90	01	C8	98	6D	53	15	85	01	8A	18	6D	B2	17	85	02
1870	90	02	E6	03	A5	03	6D	B3	17	85	03	AE	B1	17	A0	FF
1880	B1	00	91	02	88	D0	F9	B1	00	91	02	C6	01	C6	03	CA
1890	D0	EC	20	A4	18	AC	B0	17	B1	00	91	02	88	C0	FF	D0
18A0	F7	4C	11	18	AD	52	15	85	00	AD	53	15	85	01	AD	B2
18B0	17	85	02	AD	B3	17	85	03	60	20	08	14	20	E4	14	7F
18C0	0D	0A	53	45	54	20	44	45	53	54	49	4E	41	54	49	4F
18D0	4E	20	41	4E	44	20	50	52	45	53	53	20	51	2E	FF	20
18E0	07	12	AD	05	12	8D	B2	17	AD	06	12	8D	B3	17	60	00
18F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Appendix D10:

Simple Text Editor

APPENDIX D10: A SIMPLE TEXT EDITOR

SEE CHAPTER 11 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPL
BY KEN SKIER

DUMPING $1E00-$1FFF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1E00	FF	01	20	0F	1E	20	37	1E	20	C8	1E	18	18	90	F6	20
1E10	08	14	20	E4	14	7F	0D	0A	0A	53	45	54	20	55	50	20
1E20	45	44	49	54	20	42	55	46	46	45	52	2E	0D	0A	0A	FF
1E30	20	E9	15	20	A0	17	60	20	C4	11	20	2B	11	AE	03	10
1E40	A0	03	20	13	11	20	2B	11	20	76	11	20	C4	11	20	5E
1E50	1E	20	D3	11	20	76	11	20	89	1E	20	D3	11	60	20	12
1E60	15	AD	03	10	4A	AA	CA	CA	20	1A	13	CA	10	FA	AD	03
1E70	10	8D	00	1E	20	94	12	20	9B	11	20	7F	11	20	0D	13
1E80	CE	00	1E	10	EF	20	2B	15	60	AD	03	10	4A	E9	02	20
1E90	81	11	AD	01	1E	C9	01	D0	05	A9	49	18	90	02	A9	4F
1EA0	20	9B	11	A9	02	20	81	11	AD	07	10	20	9B	11	A9	02
1EB0	20	81	11	AD	06	12	20	A3	11	AD	05	12	20	A3	11	60
1EC0	06	03	3E	3C	10	7F	51	00	20	E0	12	CD	C6	1E	D0	17
1ED0	48	20	E0	12	CD	C6	1E	D0	04	68	68	68	60	8D	C7	1E
1EE0	68	20	E7	1E	AD	C7	1E	CD	C1	1E	D0	0B	CE	01	1E	10
1EF0	05	A9	01	8D	01	1E	60	CD	C2	1E	D0	04	20	79	1F	60
1F00	CD	C3	1E	D0	04	20	87	1F	60	CD	C5	1E	D0	04	20	DD
1F10	1F	60	CD	C4	1E	D0	04	20	C5	1F	60	CD	C0	1E	D0	04
1F20	20	B4	1F	60	AE	01	1E	F0	04	20	34	1F	60	20	2D	13
1F30	20	83	17	60	48	20	12	15	AD	53	15	48	AD	52	15	48
1F40	AD	55	15	48	AD	54	15	48	20	67	16	20	83	17	30	11
1F50	20	E2	18	AD	54	15	D0	04	CE	55	15	CE	54	15	20	D6
1F60	17	68	8D	54	15	68	8D	55	15	68	8D	52	15	68	8D	53
1F70	15	20	2B	15	68	20	2D	1F	60	20	94	12	C9	FF	F0	04
1F80	20	83	17	60	A9	FF	60	38	AD	53	15	CD	06	12	90	0C
1F90	D0	10	AD	52	15	CD	05	12	F0	17	B0	06	20	1A	13	A9
1FA0	00	60	AD	52	15	8D	05	12	AD	53	15	8D	06	12	A9	00
1FB0	60	A9	FF	60	20	A0	17	A9	FF	20	2D	13	20	83	17	10
1FC0	F6	20	A0	17	60	20	A0	17	20	14	14	20	94	12	C9	FF
1FD0	F0	08	20	40	14	20	83	17	10	F1	4C	1A	14	20	12	15
1FE0	AD	53	15	48	AD	52	15	48	20	E2	18	20	83	17	20	67
1FF0	16	20	D6	17	68	8D	52	15	68	8D	53	15	20	2B	15	60

Appendix D11:

Extending the Visible Monitor

APPENDIX D11:   EXTENDING THE VISIBLE MONITOR

SEE CHAPTER 12 OF BEYOND GAMES: SYSTEM SOFTWARE FOR YOUR 6502 PERSONAL COMPUTER.

DUMPING $10B0-$10FF

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
10B0	C9	50	D0	09	AD	00	14	49	FF	8D	00	14	60	C9	55	D0
10C0	09	AD	02	14	49	FF	8D	02	14	60	C9	48	D0	0D	AD	00
10D0	14	D0	04	20	57	15	60	20	AE	15	60	C9	4D	D0	04	20
10E0	B4	17	60	C9	3F	D0	0D	AD	00	14	D0	04	20	09	19	60
10F0	20	26	19	60	C9	54	D0	04	20	02	1E	60	60	00	00	00

Appendix E1:

Screen Utilities

```
APPENDIX E1          SCREEN UTILITIES


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4352 TO 4607
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


 1000 DATA  4352, 32, 196, 17, 32, 43, 17, 174, 3, 4866
 1001 DATA  4360, 16, 172, 4, 16, 32, 19, 17, 32, 4668
 1002 DATA  4368, 211, 17, 96, 142, 42, 17, 152, 170, 5215
 1003 DATA  4376, 173, 6, 16, 172, 42, 17, 145, 0, 4947
 1004 DATA  4384, 136, 16, 251, 32, 118, 17, 202, 16, 5172
 1005 DATA  4392, 239, 96, 25, 162, 0, 160, 0, 24, 5098
 1006 DATA  4400, 144, 10, 173, 4, 16, 74, 168, 173, 5162
 1007 DATA  4408, 3, 16, 74, 170, 56, 236, 3, 16, 4982
 1008 DATA  4416, 144, 3, 174, 3, 16, 56, 204, 4, 5020
 1009 DATA  4424, 16, 144, 3, 172, 4, 16, 173, 0, 4952
 1010 DATA  4432, 16, 133, 0, 173, 1, 16, 133, 1, 4905
 1011 DATA  4440, 8, 216, 138, 24, 101, 0, 144, 3, 5074
 1012 DATA  4448, 230, 1, 24, 192, 0, 240, 11, 24, 5170
 1013 DATA  4456, 109, 2, 16, 144, 2, 230, 1, 136, 5096
 1014 DATA  4464, 208, 245, 133, 0, 40, 96, 173, 2, 5361
 1015 DATA  4472, 16, 24, 144, 5, 32, 155, 17, 169, 5034
 1016 DATA  4480, 1, 8, 216, 24, 101, 0, 144, 2, 4976
 1017 DATA  4488, 230, 1, 133, 0, 56, 173, 5, 16, 5102
 1018 DATA  4496, 197, 1, 176, 5, 173, 1, 16, 133, 5198
 1019 DATA  4504, 1, 40, 96, 32, 17, 16, 160, 0, 4866
 1020 DATA  4512, 145, 0, 96, 72, 74, 74, 74, 74, 5121
 1021 DATA  4520, 32, 182, 17, 32, 124, 17, 104, 32, 5060
 1022 DATA  4528, 182, 17, 32, 124, 17, 96, 8, 216, 5220
 1023 DATA  4536, 41, 15, 201, 10, 48, 2, 105, 6, 4964
 1024 DATA  4544, 105, 48, 40, 96, 104, 170, 104, 168, 5379
 1025 DATA  4552, 165, 1, 72, 165, 0, 72, 152, 72, 5251
 1026 DATA  4560, 138, 72, 96, 104, 170, 104, 168, 104, 5516
 1027 DATA  4568, 133, 0, 104, 133, 1, 152, 72, 138, 5301
 1028 DATA  4576, 72, 96, 0, 0, 0, 0, 0, 0, 4744
```

```
1029 DATA  4584, 0, 0, 0, 0, 0, 0, 0, 0, 4584
1030 DATA  4592, 0, 0, 0, 0, 0, 0, 0, 0, 4592
1031 DATA  4600, 0, 0, 0, 0, 0, 0, 0, 0, 4600
1032 END

OK
```

Appendix E2:

Visible Monitor (Top Level and Display Subroutines)

```
APPENDIX E2       VISIBLE MONITOR (TOP LEVEL & DISPLAY SUBS)


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4608 TO 4831
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1100 DATA  4608, 0, 12, 0, 0, 49, 177, 252, 8, 5106
1101 DATA  4616, 216, 32, 18, 18, 32, 227, 18, 24, 5201
1102 DATA  4624, 144, 246, 32, 196, 17, 32, 37, 18, 5346
1103 DATA  4632, 32, 52, 18, 32, 92, 18, 32, 175, 5063
1104 DATA  4640, 18, 32, 211, 17, 96, 162, 2, 160, 5338
1105 DATA  4648, 2, 32, 60, 17, 162, 25, 160, 3, 5109
1106 DATA  4656, 32, 19, 17, 96, 162, 13, 160, 2, 5157
1107 DATA  4664, 32, 60, 17, 160, 0, 140, 81, 18, 5172
1108 DATA  4672, 185, 82, 18, 32, 124, 17, 238, 81, 5449
1109 DATA  4680, 18, 172, 81, 18, 192, 10, 208, 240, 5619
1110 DATA  4688, 96, 10, 65, 32, 32, 88, 32, 32, 5075
1111 DATA  4696, 89, 32, 32, 80, 162, 2, 160, 3, 5256
1112 DATA  4704, 32, 60, 17, 173, 6, 18, 32, 163, 5205
1113 DATA  4712, 17, 173, 5, 18, 32, 163, 17, 32, 5169
1114 DATA  4720, 127, 17, 32, 148, 18, 72, 32, 163, 5329
1115 DATA  4728, 17, 32, 127, 17, 104, 32, 124, 17, 5198
1116 DATA  4736, 32, 127, 17, 162, 0, 189, 1, 18, 5282
1117 DATA  4744, 32, 163, 17, 32, 127, 17, 232, 224, 5588
1118 DATA  4752, 4, 208, 242, 96, 165, 2, 72, 166, 5707
1119 DATA  4760, 3, 173, 5, 18, 133, 2, 173, 6, 5273
1120 DATA  4768, 18, 133, 3, 160, 0, 177, 2, 168, 5429
1121 DATA  4776, 104, 133, 2, 134, 3, 152, 96, 162, 5562
1122 DATA  4784, 2, 160, 4, 32, 60, 17, 172, 0, 5231
1123 DATA  4792, 18, 56, 192, 7, 144, 5, 160, 0, 5374
1124 DATA  4800, 140, 0, 18, 185, 205, 18, 168, 173, 5707
1125 DATA  4808, 7, 16, 145, 0, 96, 3, 6, 8, 5089
```

```
1126 DATA  4816, 11, 14, 17, 20, 0, 0, 0, 0, 4878
1127 DATA  4824, 0, 0, 0, 0, 0, 0, 0, 0, 4824
1128 END
```

Appendix E3:

Visible Monitor (Update Subroutine)

```
APPENDIX E3          VISIBLE MONITOR (UPDATE SUBROUTINE)


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4832 TO 5119
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1200 DATA  4832, 108, 8, 16, 32, 224, 18, 201, 62, 5501
1201 DATA  4840, 208, 16, 238, 0, 18, 173, 0, 18, 5511
1202 DATA  4848, 201, 7, 208, 5, 169, 0, 141, 0, 5579
1203 DATA  4856, 18, 96, 201, 60, 208, 11, 206, 0, 5656
1204 DATA  4864, 18, 16, 5, 169, 6, 141, 0, 18, 5237
1205 DATA  4872, 96, 201, 32, 208, 9, 238, 5, 18, 5679
1206 DATA  4880, 208, 3, 238, 6, 18, 96, 201, 13, 5663
1207 DATA  4888, 208, 12, 173, 5, 18, 208, 3, 206, 5721
1208 DATA  4896, 6, 18, 206, 5, 18, 96, 174, 0, 5419
1209 DATA  4904, 18, 224, 2, 208, 27, 168, 165, 0, 5716
1210 DATA  4912, 72, 166, 1, 173, 5, 18, 133, 0, 5480
1211 DATA  4920, 173, 6, 18, 133, 1, 152, 160, 0, 5563
1212 DATA  4928, 145, 0, 134, 1, 104, 133, 0, 96, 5541
1213 DATA  4936, 201, 71, 208, 35, 172, 3, 18, 174, 5818
1214 DATA  4944, 2, 18, 173, 4, 18, 72, 173, 1, 5405
1215 DATA  4952, 18, 40, 32, 108, 19, 8, 141, 1, 5319
1216 DATA  4960, 18, 142, 2, 18, 140, 3, 18, 104, 5405
1217 DATA  4968, 141, 4, 18, 96, 109, 5, 18, 72, 5430
1218 DATA  4976, 32, 213, 19, 48, 75, 168, 104, 152, 5787
1219 DATA  4984, 174, 0, 18, 208, 20, 162, 3, 24, 5593
1220 DATA  4992, 14, 5, 18, 46, 6, 18, 202, 16, 5317
1221 DATA  5000, 246, 152, 13, 5, 18, 141, 5, 18, 5598
1222 DATA  5008, 96, 224, 1, 208, 24, 41, 15, 72, 5689
1223 DATA  5016, 32, 148, 18, 10, 10, 10, 10, 41, 5295
1224 DATA  5024, 240, 141, 172, 19, 104, 13, 172, 19, 5904
1225 DATA  5032, 32, 45, 19, 96, 16, 202, 202, 202, 5846
1226 DATA  5040, 160, 3, 24, 30, 1, 18, 136, 16, 5428
1227 DATA  5048, 249, 29, 1, 18, 157, 1, 18, 96, 5617
1228 DATA  5056, 104, 201, 127, 208, 4, 32, 0, 17, 5749
```

```
1229 DATA  5064, 96, 201, 81, 208, 4, 104, 104, 40, 5902
1230 DATA  5072, 96, 32, 16, 16, 96, 56, 233, 48, 5665
1231 DATA  5080, 144, 15, 201, 10, 144, 14, 233, 7, 5848
1232 DATA  5088, 201, 16, 176, 5, 56, 201, 10, 176, 5929
1233 DATA  5096, 3, 169, 255, 96, 162, 0, 96, 0, 5877
1234 DATA  5104, 0, 0, 0, 0, 0, 0, 0, 0, 5104
1235 DATA  5112, 0, 0, 0, 0, 0, 0, 0, 0, 5112
1236 END
```

Appendix E4:

Print Utilities

```
APPENDIX E4        PRINT UTILITIES


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  5120 TO 5455
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1300 DATA  5120, 0, 255, 0, 0, 0, 0, 0, 0, 5375
1301 DATA  5128, 169, 255, 141, 1, 20, 96, 169, 0, 5979
1302 DATA  5136, 141, 1, 20, 96, 169, 255, 141, 0, 5959
1303 DATA  5144, 20, 96, 169, 0, 141, 0, 20, 96, 5686
1304 DATA  5152, 169, 255, 141, 2, 20, 96, 169, 0, 6004
1305 DATA  5160, 141, 2, 20, 96, 32, 8, 20, 32, 5511
1306 DATA  5168, 20, 20, 32, 32, 20, 96, 32, 14, 5434
1307 DATA  5176, 20, 32, 26, 20, 32, 38, 20, 96, 5460
1308 DATA  5184, 201, 0, 240, 36, 141, 3, 20, 173, 5998
1309 DATA  5192, 1, 20, 240, 6, 173, 3, 20, 32, 5687
1310 DATA  5200, 105, 20, 173, 0, 20, 240, 6, 173, 5937
1311 DATA  5208, 3, 20, 32, 108, 20, 173, 2, 20, 5586
1312 DATA  5216, 240, 6, 173, 3, 20, 32, 111, 20, 5821
1313 DATA  5224, 96, 108, 10, 16, 108, 12, 16, 108, 5698
1314 DATA  5232, 14, 16, 169, 13, 32, 64, 20, 169, 5729
1315 DATA  5240, 10, 32, 64, 20, 96, 169, 32, 32, 5695
1316 DATA  5248, 64, 20, 96, 72, 74, 74, 74, 74, 5796
1317 DATA  5256, 32, 182, 17, 32, 64, 20, 104, 32, 5739
1318 DATA  5264, 182, 17, 32, 64, 20, 96, 169, 32, 5876
1319 DATA  5272, 142, 4, 20, 72, 174, 4, 20, 240, 5948
1320 DATA  5280, 10, 206, 4, 20, 32, 64, 20, 104, 5740
1321 DATA  5288, 24, 144, 240, 104, 96, 142, 4, 20, 6062
1322 DATA  5296, 174, 4, 20, 240, 9, 206, 4, 20, 5973
1323 DATA  5304, 32, 114, 20, 24, 144, 242, 96, 142, 6118
1324 DATA  5312, 5, 20, 181, 1, 72, 181, 0, 72, 5844
1325 DATA  5320, 174, 5, 20, 161, 0, 201, 255, 240, 6376
1326 DATA  5328, 12, 246, 0, 208, 2, 246, 1, 32, 6075
1327 DATA  5336, 64, 20, 24, 144, 235, 104, 149, 0, 6076
1328 DATA  5344, 104, 149, 1, 96, 104, 170, 104, 168, 6240
```

```
1329 DATA  5352, 32, 18, 21, 142, 5, 18, 140, 6, 5734
1330 DATA  5360, 18, 32, 13, 19, 32, 13, 19, 32, 5538
1331 DATA  5368, 148, 18, 201, 255, 240, 6, 32, 64, 6332
1332 DATA  5376, 20, 24, 144, 240, 174, 5, 18, 172, 6173
1333 DATA  5384, 6, 18, 32, 43, 21, 152, 72, 138, 5866
1334 DATA  5392, 72, 96, 104, 141, 6, 20, 104, 141, 6076
1335 DATA  5400, 7, 20, 173, 6, 18, 72, 173, 5, 5874
1336 DATA  5408, 18, 72, 173, 7, 20, 72, 173, 6, 5949
1337 DATA  5416, 20, 72, 96, 104, 141, 6, 20, 104, 5979
1338 DATA  5424, 141, 7, 20, 104, 141, 5, 18, 104, 5964
1339 DATA  5432, 141, 6, 18, 173, 7, 20, 72, 173, 6042
1340 DATA  5440, 6, 20, 72, 96, 0, 0, 0, 0, 5634
1341 DATA  5448, 0, 0, 0, 0, 0, 0, 0, 0, 5448
1342 END
```

Appendix E5:

Two Hexdump Tools

```
APPENDIX E5        TWO HEXDUMP TOOLS


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  5456 TO 6063
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


 1400 DATA  5456, 0, 4, 0, 0, 255, 255, 0, 32, 6002
 1401 DATA  5464, 8, 20, 173, 81, 21, 141, 80, 21, 6009
 1402 DATA  5472, 173, 5, 18, 41, 248, 141, 5, 18, 6121
 1403 DATA  5480, 32, 114, 20, 32, 114, 20, 32, 161, 6005
 1404 DATA  5488, 21, 32, 114, 20, 32, 125, 20, 32, 5884
 1405 DATA  5496, 154, 21, 32, 13, 19, 173, 5, 18, 5931
 1406 DATA  5504, 41, 7, 208, 240, 32, 114, 20, 173, 6339
 1407 DATA  5512, 5, 18, 41, 15, 208, 3, 32, 114, 5948
 1408 DATA  5520, 20, 206, 80, 21, 208, 216, 32, 14, 6317
 1409 DATA  5528, 20, 96, 32, 148, 18, 32, 131, 20, 6025
 1410 DATA  5536, 96, 173, 6, 18, 32, 131, 20, 173, 6185
 1411 DATA  5544, 5, 18, 32, 131, 20, 96, 32, 201, 6079
 1412 DATA  5552, 21, 32, 233, 21, 32, 160, 23, 32, 6106
 1413 DATA  5560, 20, 20, 32, 235, 22, 32, 66, 23, 6010
 1414 DATA  5568, 16, 251, 32, 114, 20, 32, 26, 20, 6079
 1415 DATA  5576, 96, 32, 0, 17, 32, 8, 20, 32, 5813
 1416 DATA  5584, 228, 20, 127, 13, 80, 82, 73, 78, 6285
 1417 DATA  5592, 84, 73, 78, 71, 32, 72, 69, 88, 6159
 1418 DATA  5600, 68, 85, 77, 80, 13, 10, 10, 255, 6198
 1419 DATA  5608, 96, 32, 8, 20, 32, 228, 20, 127, 6171
 1420 DATA  5616, 13, 10, 83, 69, 84, 32, 83, 84, 6074
 1421 DATA  5624, 65, 82, 84, 73, 78, 71, 32, 65, 6174
 1422 DATA  5632, 68, 68, 82, 69, 83, 83, 32, 65, 6182
 1423 DATA  5640, 78, 68, 32, 80, 82, 69, 83, 83, 6215
 1424 DATA  5648, 32, 34, 81, 34, 46, 255, 32, 7, 6169
 1425 DATA  5656, 18, 32, 103, 22, 32, 8, 20, 32, 5923
 1426 DATA  5664, 228, 20, 127, 13, 10, 83, 69, 84, 6298
 1427 DATA  5672, 32, 69, 78, 68, 32, 65, 68, 68, 6152
 1428 DATA  5680, 82, 69, 83, 83, 32, 65, 78, 68, 6240
```

```
1429 DATA  5688, 32, 80, 82, 69, 83, 83, 32, 34, 6183
1430 DATA  5696, 81, 34, 46, 255, 32, 7, 18, 56, 6225
1431 DATA  5704, 173, 6, 18, 205, 83, 21, 144, 36, 6390
1432 DATA  5712, 208, 8, 173, 5, 18, 205, 82, 21, 6432
1433 DATA  5720, 144, 26, 173, 6, 18, 141, 85, 21, 6334
1434 DATA  5728, 173, 5, 18, 141, 84, 21, 96, 173, 6439
1435 DATA  5736, 6, 18, 141, 83, 21, 173, 5, 18, 6201
1436 DATA  5744, 141, 82, 21, 96, 32, 228, 20, 127, 6491
1437 DATA  5752, 13, 10, 10, 10, 32, 69, 82, 82, 6060
1438 DATA  5760, 79, 82, 33, 33, 33, 32, 69, 78, 6199
1439 DATA  5768, 68, 32, 65, 68, 68, 82, 69, 83, 6303
1440 DATA  5776, 83, 32, 76, 69, 83, 83, 32, 84, 6318
1441 DATA  5784, 72, 65, 78, 32, 83, 84, 65, 82, 6345
1442 DATA  5792, 84, 32, 65, 68, 68, 82, 69, 83, 6343
1443 DATA  5800, 83, 44, 32, 87, 72, 73, 67, 72, 6330
1444 DATA  5808, 32, 73, 83, 32, 255, 32, 187, 22, 6524
1445 DATA  5816, 76, 28, 22, 169, 36, 32, 64, 20, 6263
1446 DATA  5824, 173, 83, 21, 32, 131, 20, 173, 82, 6539
1447 DATA  5832, 21, 32, 131, 20, 96, 169, 36, 32, 6369
1448 DATA  5840, 64, 20, 173, 85, 21, 32, 131, 20, 6386
1449 DATA  5848, 173, 64, 21, 32, 131, 20, 96, 32, 6437
1450 DATA  5856, 187, 22, 169, 45, 32, 64, 20, 32, 6427
1451 DATA  5864, 205, 22, 96, 32, 228, 20, 127, 13, 6607
1452 DATA  5872, 10, 10, 68, 85, 77, 80, 73, 78, 6353
1453 DATA  5880, 71, 32, 255, 32, 223, 22, 32, 114, 6661
1454 DATA  5888, 20, 32, 228, 20, 127, 10, 10, 32, 6367
1455 DATA  5896, 32, 32, 32, 32, 32, 32, 32, 48, 6168
1456 DATA  5904, 32, 32, 49, 32, 32, 50, 32, 32, 6195
1457 DATA  5912, 51, 32, 32, 52, 32, 32, 53, 32, 6228
1458 DATA  5920, 32, 54, 32, 32, 55, 32, 32, 56, 6245
1459 DATA  5928, 32, 32, 57, 32, 32, 65, 32, 32, 6242
1460 DATA  5936, 66, 32, 32, 67, 32, 32, 68, 32, 6297
1461 DATA  5944, 32, 69, 32, 32, 70, 13, 10, 10, 6212
1462 DATA  5952, 255, 96, 32, 114, 20, 173, 5, 18, 6665
1463 DATA  5960, 72, 41, 15, 141, 86, 21, 104, 41, 6481
1464 DATA  5968, 240, 141, 5, 18, 32, 161, 21, 162, 6748
1465 DATA  5976, 3, 32, 150, 20, 173, 86, 21, 240, 6701
1466 DATA  5984, 13, 162, 3, 32, 150, 20, 32, 13, 6409
1467 DATA  5992, 19, 206, 86, 21, 208, 243, 32, 154, 6961
1468 DATA  6000, 21, 32, 125, 20, 32, 131, 23, 48, 6432
1469 DATA  6008, 9, 173, 5, 18, 41, 15, 201, 0, 6470
1470 DATA  6016, 208, 236, 96, 56, 173, 6, 18, 205, 7014
1471 DATA  6024, 85, 21, 144, 11, 208, 15, 56, 173, 6737
1472 DATA  6032, 5, 18, 205, 84, 21, 176, 6, 32, 6579
1473 DATA  6040, 13, 19, 169, 0, 96, 169, 255, 96, 6857
1474 DATA  6048, 173, 82, 21, 141, 5, 18, 173, 83, 6744
1475 DATA  6056, 21, 141, 6, 18, 96, 0, 0, 0, 6338
1476 END
```

Appendix E6:

Table-Driven Disassembler (Top Level and Utility Subroutines)

```
APPENDIX E6        DISASSEMBLER (TOP LEVEL & UTILITY SUBS)


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  6400 TO 6719
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1500 DATA  6400, 5, 0, 0, 0, 0, 0, 0, 0, 6405
1501 DATA  6408, 16, 32, 8, 20, 173, 0, 25, 141, 6823
1502 DATA  6416, 1, 25, 169, 255, 141, 84, 21, 141, 7253
1503 DATA  6424, 85, 21, 32, 114, 20, 32, 125, 25, 6878
1504 DATA  6432, 206, 1, 25, 208, 248, 96, 32, 26, 7274
1505 DATA  6440, 20, 32, 8, 20, 32, 228, 20, 127, 6927
1506 DATA  6448, 13, 10, 32, 32, 32, 32, 32, 80, 6711
1507 DATA  6456, 82, 73, 78, 84, 73, 78, 71, 32, 7027
1508 DATA  6464, 68, 73, 83, 65, 83, 83, 69, 77, 7065
1509 DATA  6472, 66, 76, 69, 82, 46, 13, 10, 255, 7089
1510 DATA  6480, 32, 233, 21, 32, 20, 20, 32, 228, 7098
1511 DATA  6488, 20, 127, 13, 10, 68, 73, 83, 65, 6947
1512 DATA  6496, 83, 83, 69, 77, 66, 76, 73, 78, 7101
1513 DATA  6504, 71, 32, 255, 32, 223, 22, 32, 160, 7331
1514 DATA  6512, 23, 32, 114, 20, 32, 125, 25, 16, 6899
1515 DATA  6520, 251, 32, 26, 20, 96, 32, 148, 18, 7143
1516 DATA  6528, 72, 32, 146, 25, 32, 125, 20, 104, 7084
1517 DATA  6536, 32, 175, 25, 32, 1, 26, 32, 131, 6990
1518 DATA  6544, 23, 96, 162, 3, 142, 2, 25, 170, 7167
1519 DATA  6552, 189, 0, 28, 170, 189, 80, 27, 142, 7377
1520 DATA  6560, 3, 25, 32, 64, 20, 174, 3, 25, 6906
1521 DATA  6568, 232, 206, 2, 25, 208, 238, 96, 170, 7745
1522 DATA  6576, 189, 0, 29, 170, 32, 184, 25, 96, 7301
1523 DATA  6584, 189, 27, 27, 141, 4, 25, 232, 189, 7418
1524 DATA  6592, 27, 27, 141, 5, 25, 108, 4, 25, 6954
1525 DATA  6600, 32, 13, 19, 32, 154, 21, 96, 32, 6999
```

```
1526 DATA  6608, 13, 19, 32, 148, 18, 72, 32, 13, 6955
1527 DATA  6616, 19, 32, 154, 21, 104, 32, 131, 20, 7129
1528 DATA  6624, 96, 169, 40, 208, 2, 169, 41, 32, 7381
1529 DATA  6632, 64, 20, 96, 169, 44, 32, 64, 20, 7141
1530 DATA  6640, 169, 88, 32, 64, 20, 96, 169, 44, 7322
1531 DATA  6648, 32, 64, 20, 169, 89, 32, 64, 20, 7138
1532 DATA  6656, 96, 141, 7, 25, 142, 6, 25, 202, 7300
1533 DATA  6664, 48, 6, 32, 26, 19, 202, 16, 250, 7263
1534 DATA  6672, 8, 216, 56, 173, 8, 25, 233, 4, 7395
1535 DATA  6680, 237, 7, 25, 40, 170, 32, 150, 20, 7361
1536 DATA  6688, 32, 161, 21, 32, 125, 20, 32, 154, 7265
1537 DATA  6696, 21, 32, 13, 19, 206, 6, 25, 16, 7034
1538 DATA  6704, 242, 32, 26, 19, 32, 114, 20, 96, 7285
1539 DATA  6712, 0, 0, 0, 0, 0, 0, 0, 0, 6712
1540 END
```

Appendix E7:

Table-Driven Disassembler (Addressing Mode Subroutines)

```
APPENDIX E7        DISASSEMBLER (ADDRESSING MODE SUBROUTINES)


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  6720 TO 6991
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


 1600 DATA  6720, 32, 207, 25. 162, 2, 169, 4, 96, 7417
 1601 DATA  6728, 32, 64, 26, 32, 235, 25, 162, 2, 7306
 1602 DATA  6736, 169, 6, 96, 32, 64, 26, 32, 246, 7407
 1603 DATA  6744, 25, 162, 2, 169, 6, 96, 169, 65, 7438
 1604 DATA  6752, 32, 64, 20, 162, 0, 169, 1, 96, 7296
 1605 DATA  6760, 162, 0, 169, 0, 96, 169, 35, 32, 7423
 1606 DATA  6768, 64, 20, 169, 36, 32, 64, 20, 32, 7205
 1607 DATA  6776, 200, 25, 162, 1, 169, 4, 96, 32, 7465
 1608 DATA  6784, 225, 25, 32, 64, 26, 32, 229, 25, 7442
 1609 DATA  6792, 169, 6, 162, 2, 96, 32, 225, 25, 7509
 1610 DATA  6800. 32, 232, 26, 32, 229, 25, 162, 1, 7539
 1611 DATA  6808, 169, 8, 96, 32, 225, 25, 32, 219, 7614
 1612 DATA  6816, 26, 32. 229, 25. 32, 246, 25, 162, 7593
 1613 DATA  6824, 1, 169, 8, 96, 32, 13, 19, 32, 7194
 1614 DATA  6832, 18, 21. 32. 148, 18, 72, 32, 13, 7186
 1615 DATA  6840, 19, 104, 201, 0, 16, 3, 206, 6, 7395
 1616 DATA  6848. 18, 8, 216, 24, 109, 5, 18, 144, 7390
 1617 DATA  6856, 3, 238, 6, 18, 141, 5, 18, 40, 7325
 1618 DATA  6864, 32, 161, 21, 32, 43, 21, 162, 1, 7337
 1619 DATA  6872, 169, 4, 96, 169, 0, 32, 131, 20, 7493
 1620 DATA  6880, 32, 200, 25, 162, 1, 169, 4, 96, 7569
 1621 DATA  6888, 32, 219, 26, 32, 235, 25, 162, 1, 7620
 1622 DATA  6896, 169, 6, 96, 32, 219, 26, 32, 246, 7722
 1623 DATA  6904, 25, 162, 1, 169, 6, 96, 104, 104, 7571
 1624 DATA  6912, 104, 104, 32, 131, 23, 48, 13, 32, 7399
 1625 DATA  6920, 148, 18, 201, 255, 240, 6, 32, 64, 7884
```

```
1626 DATA  6928, 20, 24, 144, 238, 32, 114, 20, 32, 7552
1627 DATA  6936, 131, 23, 96, 104, 26, 94, 26, 109, 7545
1628 DATA  6944, 26, 219, 26, 232, 26, 243, 26, 64, 7806
1629 DATA  6952, 26, 72, 26, 83, 26, 104, 26, 172, 7487
1630 DATA  6960, 26, 141, 26, 155, 26, 127, 26, 254, 7741
1631 DATA  6968, 26, 0, 0, 0, 0, 0, 0, 0, 6994
1632 DATA  6976, 0, 0, 0, 0, 0, 0, 0, 0, 6976
1633 DATA  6984, 0, 0, 0, 0, 0, 0, 0, 0, 6984
1634 END
```

Appendix E8:

Table-Driven Disassembler (Tables)

```
APPENDIX E8        DISASSEMBLER (TABLES)


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  6992 TO 7679
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1700 DATA  6992, 127, 66, 65, 68, 65, 68, 67, 65, 7583
1701 DATA  7000, 78, 68, 65, 83, 76, 66, 67, 67, 7570
1702 DATA  7008, 66, 67, 83, 66, 69, 81, 66, 73, 7579
1703 DATA  7016, 84, 66, 77, 73, 66, 78, 69, 66, 7595
1704 DATA  7024, 80, 76, 66, 82, 75, 66, 86, 67, 7622
1705 DATA  7032, 66, 86, 83, 67, 76, 67, 67, 76, 7620
1706 DATA  7040, 68, 67, 76, 73, 67, 76, 86, 67, 7620
1707 DATA  7048, 77, 80, 67, 80, 88, 67, 80, 89, 7676
1708 DATA  7056, 68, 69, 67, 68, 69, 88, 68, 69, 7622
1709 DATA  7064, 89, 69, 79, 82, 73, 78, 67, 73, 7674
1710 DATA  7072, 78, 88, 73, 78, 89, 74, 77, 80, 7709
1711 DATA  7080, 74, 83, 82, 76, 68, 65, 76, 68, 7672
1712 DATA  7088, 88, 76, 68, 89, 76, 83, 82, 78, 7728
1713 DATA  7096, 79, 80, 79, 82, 65, 80, 72, 65, 7698
1714 DATA  7104, 80, 72, 80, 80, 76, 65, 80, 76, 7713
1715 DATA  7112, 80, 82, 79, 76, 82, 79, 82, 82, 7754
1716 DATA  7120, 84, 73, 82, 84, 83, 83, 66, 67, 7742
1717 DATA  7128, 83, 69, 67, 83, 69, 68, 83, 69, 7719
1718 DATA  7136, 73, 83, 84, 65, 83, 84, 88, 83, 7779
1719 DATA  7144, 84, 89, 84, 65, 88, 84, 65, 89, 7792
1720 DATA  7152, 84, 83, 88, 84, 88, 65, 84, 88, 7816
1721 DATA  7160, 83, 84, 89, 65, 84, 69, 88, 255, 7977
1722 DATA  7168, 34, 106, 1, 1, 1, 106, 10, 1, 7428
1723 DATA  7176, 112, 106, 10, 1, 1, 106, 10, 1, 7523
1724 DATA  7184, 31, 106, 1, 1, 1, 106, 10, 1, 7441
1725 DATA  7192, 43, 106, 1, 1, 1, 106, 10, 1, 7461
1726 DATA  7200, 88, 7, 1, 1, 22, 7, 121, 1, 7448
1727 DATA  7208, 118, 7, 121, 1, 22, 7, 121, 1, 7606
1728 DATA  7216, 25, 7, 1, 1, 1, 7, 121, 1, 7380
```

```
1729 DATA 7224, 136, 7, 1, 1, 1, 7, 121, 1, 7499
1730 DATA 7232, 127, 73, 1, 1, 1, 73, 100, 1, 7609
1731 DATA 7240, 109, 73, 100, 1, 85, 73, 100, 1, 7782
1732 DATA 7248, 37, 73, 1, 1, 1, 73, 100, 1, 7535
1733 DATA 7256, 49, 73, 1, 1, 1, 73, 100, 1, 7555
1734 DATA 7264, 130, 4, 1, 1, 1, 4, 124, 1, 7530
1735 DATA 7272, 115, 4, 124, 1, 85, 4, 124, 1, 7730
1736 DATA 7280, 40, 4, 1, 1, 1, 4, 124, 1, 7456
1737 DATA 7288, 142, 4, 1, 1, 1, 4, 124, 172, 7737
1738 DATA 7296, 1, 145, 1, 1, 151, 145, 148, 1, 7889
1739 DATA 7304, 70, 1, 163, 1, 151, 145, 148, 1, 7984
1740 DATA 7312, 13, 145, 1, 1, 151, 145, 148, 1, 7917
1741 DATA 7320, 169, 145, 163, 1, 1, 145, 1, 1, 7946
1742 DATA 7328, 97, 91, 94, 1, 97, 91, 94, 1, 7894
1743 DATA 7336, 157, 91, 154, 1, 97, 91, 94, 1, 8022
1744 DATA 7344, 16, 91, 1, 1, 97, 91, 94, 1, 7736
1745 DATA 7352, 52, 91, 158, 1, 97, 91, 94, 1, 7937
1746 DATA 7360, 61, 55, 1, 1, 61, 55, 64, 1, 7659
1747 DATA 7368, 82, 55, 67, 1, 61, 55, 64, 1, 7754
1748 DATA 7376, 28, 55, 1, 1, 1, 55, 64, 1, 7582
1749 DATA 7384, 46, 55, 1, 1, 1, 55, 64, 1, 7608
1750 DATA 7392, 58, 133, 1, 1, 58, 133, 76, 1, 7853
1751 DATA 7400, 79, 133, 103, 1, 58, 133, 76, 1, 7984
1752 DATA 7408, 19, 133, 1, 1, 1, 133, 76, 1, 7773
1753 DATA 7416, 139, 133, 1, 1, 1, 133, 76, 1, 7901
1754 DATA 7424, 18, 22, 0, 0, 0, 6, 6, 0, 7476
1755 DATA 7432, 18, 4, 2, 0, 0, 12, 12, 0, 7480
1756 DATA 7440, 20, 24, 0, 0, 0, 14, 14, 0, 7512
1757 DATA 7448, 18, 16, 0, 0, 0, 22, 22, 0, 7526
1758 DATA 7456, 12, 22, 0, 0, 6, 6, 6, 0, 7508
1759 DATA 7464, 18, 4, 2, 0, 12, 12, 12, 0, 7524
1760 DATA 7472, 20, 24, 0, 0, 0, 8, 8, 0, 7532
1761 DATA 7480, 18, 16, 0, 0, 0, 14, 14, 0, 7542
1762 DATA 7488, 18, 22, 0, 0, 0, 6, 6, 0, 7540
1763 DATA 7496, 18, 12, 2, 0, 12, 12, 12, 0, 7564
1764 DATA 7504, 20, 24, 0, 0, 0, 8, 8, 0, 7564
1765 DATA 7512, 18, 16, 0, 0, 0, 14, 14, 0, 7574
1766 DATA 7520, 18, 22, 0, 0, 0, 6, 6, 0, 7572
1767 DATA 7528, 18, 4, 2, 0, 26, 12, 12, 0, 7602
1768 DATA 7536, 20, 24, 0, 0, 0, 8, 8, 0, 7596
1769 DATA 7544, 18, 16, 0, 0, 0, 14, 14, 28, 7634
1770 DATA 7552, 0, 22, 0, 0, 6, 6, 6, 0, 7592
1771 DATA 7560, 18, 0, 18, 0, 12, 12, 12, 0, 7632
1772 DATA 7568, 20, 24, 0, 0, 8, 8, 10, 0, 7638
1773 DATA 7576, 18, 16, 18, 0, 0, 14, 0, 0, 7642
1774 DATA 7584, 4, 22, 4, 0, 6, 6, 6, 0, 7632
1775 DATA 7592, 18, 4, 18, 0, 12, 12, 12, 0, 7668
1776 DATA 7600, 20, 24, 0, 0, 8, 8, 10, 0, 7670
1777 DATA 7608, 20, 16, 18, 0, 14, 14, 16, 0, 7706
1778 DATA 7616, 4, 22, 0, 0, 6, 6, 6, 0, 7660
1779 DATA 7624, 18, 4, 18, 0, 12, 12, 12, 0, 7700
1780 DATA 7632, 20, 24, 0, 0, 0, 8, 8, 0, 7692
1781 DATA 7640, 18, 16, 0, 0, 0, 14, 14, 0, 7702
1782 DATA 7648, 4, 22, 0, 0, 6, 6, 6, 0, 7692
1783 DATA 7656, 18, 4, 18, 0, 12, 12, 12, 0, 7732
1784 DATA 7664, 20, 24, 0, 0, 0, 8, 8, 0, 7724
1785 DATA 7672, 18, 16, 0, 0, 0, 14, 14, 0, 7734
1786 END
```

Appendix E9:

Move Utilities

```
APPENDIX E9        MOVE UTILITIES


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  6064 TO 6399
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1800 DATA  6064, 0, 0, 0, 0, 32, 8, 20, 32, 6156
1801 DATA  6072, 228, 20, 127, 13, 10, 32, 32, 32, 6566
1802 DATA  6080, 32, 32, 77, 79, 86, 69, 32, 84, 6571
1803 DATA  6088, 79, 79, 76, 46, 13, 10, 10, 255, 6656
1804 DATA  6096, 32, 233, 21, 32, 185, 24, 174, 85, 6882
1805 DATA  6104, 21, 56, 173, 84, 21, 237, 82, 21, 6799
1806 DATA  6112, 141, 176, 23, 176, 2, 202, 56, 138, 7026
1807 DATA  6120, 237, 83, 21, 141, 177, 23, 176, 3, 6981
1808 DATA  6128, 169, 0, 96, 160, 3, 185, 0, 0, 6741
1809 DATA  6136, 72, 136, 16, 249, 56, 173, 83, 21, 6942
1810 DATA  6144, 205, 179, 23, 144, 64, 208, 24, 173, 7164
1811 DATA  6152, 82, 21, 205, 178, 23, 144, 54, 208, 7067
1812 DATA  6160, 14, 160, 0, 104, 153, 0, 0, 200, 6791
1813 DATA  6168, 192, 4, 208, 247, 169, 255, 96, 32, 7371
1814 DATA  6176, 164, 24, 160, 0, 174, 177, 23, 240, 7138
1815 DATA  6184, 14, 177, 0, 145, 2, 200, 208, 249, 7179
1816 DATA  6192, 230, 1, 230, 3, 202, 208, 242, 136, 7444
1817 DATA  6200, 200, 177, 0, 145, 2, 204, 176, 23, 7127
1818 DATA  6208, 208, 246, 76, 17, 24, 173, 177, 23, 7152
1819 DATA  6216, 240, 72, 172, 177, 23, 173, 176, 23, 7272
1820 DATA  6224, 56, 233, 255, 176, 1, 136, 170, 132, 7383
1821 DATA  6232, 3, 138, 24, 109, 82, 21, 133, 0, 6742
1822 DATA  6240, 144, 1, 200, 152, 109, 83, 21, 133, 7083
1823 DATA  6248, 1, 138, 24, 109, 178, 23, 133, 2, 6856
1824 DATA  6256, 144, 2, 230, 3, 165, 3, 109, 179, 7091
1825 DATA  6264, 23, 133, 3, 174, 177, 23, 160, 255, 7212
1826 DATA  6272, 177, 0, 145, 2, 136, 208, 249, 177, 7366
1827 DATA  6280, 0, 145, 2, 198, 1, 198, 3, 202, 7029
1828 DATA  6288, 208, 236, 32, 164, 24, 172, 176, 23, 7323
```

```
1829 DATA  6296, 177, 0, 145, 2, 136, 192, 255, 208, 7411
1830 DATA  6304, 247, 76, 17, 24, 173, 82, 21, 133, 7077
1831 DATA  6312, 0, 173, 83, 21, 133, 1, 173, 178, 7074
1832 DATA  6320, 23, 133, 2, 173, 179, 23, 133, 3, 6989
1833 DATA  6328, 96, 32, 8, 20, 32, 228, 20, 127, 6891
1834 DATA  6336, 13, 10, 83, 69, 84, 32, 68, 69, 6764
1835 DATA  6344, 83, 84, 73, 78, 65, 84, 73, 79, 6963
1836 DATA  6352, 78, 32, 65, 78, 68, 32, 80, 82, 6867
1837 DATA  6360, 69, 83, 83, 32, 81, 46, 255, 32, 7041
1838 DATA  6368, 7, 18, 173, 5, 18, 141, 178, 23, 6931
1839 DATA  6376, 173, 6, 18, 141, 179, 23, 96, 0, 7012
1840 DATA  6384, 0, 0, 0, 0, 0, 0, 0, 0, 6384
1841 DATA  6392, 0, 0, 0, 0, 0, 0, 0, 0, 6392
1842 END
```

Appendix E10:

Simple Text Editor

```
APPENDIX E10        A SIMPLE TEXT EDITOR


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  7680 TO 8191
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


1900 DATA  7680, 255, 1, 32, 15, 30, 32, 55, 30, 8130
1901 DATA  7688, 32, 200, 30, 24, 24, 144, 246, 32, 8420
1902 DATA  7696, 8, 20, 32, 228, 20, 127, 13, 10, 8154
1903 DATA  7704, 10, 83, 69, 84, 32, 85, 80, 32, 8179
1904 DATA  7712, 69, 68, 73, 84, 32, 66, 85, 70, 8259
1905 DATA  7720, 70, 69, 82, 46, 13, 10, 10, 255, 8275
1906 DATA  7728, 32, 233, 21, 32, 160, 23, 96, 32, 8357
1907 DATA  7736, 196, 17, 32, 43, 17, 174, 3, 16, 8234
1908 DATA  7744, 160, 3, 32, 19, 17, 32, 43, 17, 8067
1909 DATA  7752, 32, 118, 17, 32, 196, 17, 32, 94, 8290
1910 DATA  7760, 30, 32, 211, 17, 32, 118, 17, 32, 8249
1911 DATA  7768, 137, 30, 32, 211, 17, 96, 32, 18, 8341
1912 DATA  7776, 21, 173, 3, 16, 74, 170, 202, 202, 8637
1913 DATA  7784, 32, 26, 19, 202, 16, 250, 173, 3, 8505
1914 DATA  7792, 16, 141, 0, 30, 32, 148, 18, 32, 8209
1915 DATA  7800, 155, 17, 32, 127, 17, 32, 13, 19, 8212
1916 DATA  7808, 206, 0, 30, 16, 239, 32, 43, 21, 8395
1917 DATA  7816, 96, 173, 3, 16, 74, 233, 2, 32, 8445
1918 DATA  7824, 129, 17, 173, 1, 30, 201, 1, 208, 8584
1919 DATA  7832, 5, 169, 73, 24, 144, 2, 169, 79, 8497
1920 DATA  7840, 32, 155, 17, 169, 2, 32, 129, 17, 8393
1921 DATA  7848, 173, 7, 16, 32, 155, 17, 169, 2, 8419
1922 DATA  7856, 32, 129, 17, 173, 6, 18, 32, 163, 8426
1923 DATA  7864, 17, 173, 5, 18, 32, 163, 17, 96, 8385
1924 DATA  7872, 6, 3, 62, 60, 16, 127, 81, 0, 8227
1925 DATA  7880, 32, 224, 18, 205, 198, 30, 208, 23, 8818
1926 DATA  7888, 72, 32, 224, 18, 205, 198, 30, 208, 8875
1927 DATA  7896, 4, 104, 104, 104, 96. 141, 199, 30, 8678
1928 DATA  7904, 104, 32. 231, 30, 173, 199, 30, 205, 8908
```

```
1929 DATA 7912, 193, 30, 208, 11, 206, 1, 30, 16, 8607
1930 DATA 7920, 5, 169, 1, 141, 1, 30, 96, 205, 8568
1931 DATA 7928, 194, 30, 208, 4, 32, 121, 31, 96, 8644
1932 DATA 7936, 205, 195, 30, 208, 4, 32, 135, 31, 8776
1933 DATA 7944, 96, 205, 197, 30, 208, 4, 32, 221, 8937
1934 DATA 7952, 31, 96, 205, 196, 30, 208, 4, 32, 8754
1935 DATA 7960, 197, 31, 96, 205, 192, 30, 208, 4, 8923
1936 DATA 7968, 32, 180, 31, 96, 174, 1, 30, 240, 8752
1937 DATA 7976, 4, 32, 52, 31, 96, 32, 45, 19, 8287
1938 DATA 7984, 32, 131, 23, 96, 72, 32, 18, 21, 8409
1939 DATA 7992, 173, 83, 21, 72, 173, 82, 21, 72, 8669
1940 DATA 8000, 173, 85, 21, 72, 173, 84, 21, 72, 8701
1941 DATA 8008, 32, 103, 22, 32, 131, 23, 48, 17, 8416
1942 DATA 8016, 32, 226, 24, 173, 84, 21, 208, 4, 8788
1943 DATA 8024, 206, 85, 21, 206, 84, 21, 32, 214, 8893
1944 DATA 8032, 23, 104, 141, 84, 21, 104, 141, 85, 8735
1945 DATA 8040, 21, 104, 141, 82, 21, 104, 141, 83, 8737
1946 DATA 8048, 21, 32, 43, 21, 104, 32, 45, 31, 8377
1947 DATA 8056, 96, 32, 148, 18, 201, 255, 240, 4, 9050
1948 DATA 8064, 32, 131, 23, 96, 169, 255, 96, 56, 8922
1949 DATA 8072, 173, 83, 21, 205, 6, 18, 144, 12, 8734
1950 DATA 8080, 208, 16, 173, 82, 21, 205, 5, 18, 8808
1951 DATA 8088, 240, 23, 176, 6, 32, 26, 19, 169, 8779
1952 DATA 8096, 0, 96, 173, 82, 21, 141, 5, 18, 8632
1953 DATA 8104, 173, 83, 21, 141, 6, 18, 169, 0, 8715
1954 DATA 8112, 96, 169, 255, 96, 32, 160, 23, 169, 9112
1955 DATA 8120, 255, 32, 45, 19, 32, 131, 23, 16, 8673
1956 DATA 8128, 246, 32, 160, 23, 96, 32, 160, 23, 8900
1957 DATA 8136, 32, 20, 20, 32, 148, 18, 201, 255, 8862
1958 DATA 8144, 240, 8, 32, 64, 20, 32, 131, 23, 8694
1959 DATA 8152, 16, 241, 76, 26, 20, 32, 18, 21, 8602
1960 DATA 8160, 173, 83, 21, 72, 173, 82, 21, 72, 8857
1961 DATA 8168, 32, 226, 24, 32, 131, 23, 32, 103, 8771
1962 DATA 8176, 22, 32, 214, 23, 104, 141, 82, 21, 8815
1963 DATA 8184, 104, 141, 83, 21, 32, 43, 21, 96, 8725
1964 END
```

Appendix E11:

Extending the Visible Monitor

```
APPENDIX E11        EXTENDING THE VISIBLE MONITOR


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4272 TO 4351
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


 2000 DATA  4272, 201, 80, 208, 9, 173, 0, 20, 73, 5036
 2001 DATA  4280, 255, 141, 0, 20, 96, 201, 85, 208, 5286
 2002 DATA  4288, 9, 173, 2, 20, 73, 255, 141, 2, 4963
 2003 DATA  4296, 20, 96, 201, 72, 208, 13, 173, 0, 5079
 2004 DATA  4304, 20, 208, 4, 32, 87, 21, 96, 32, 4804
 2005 DATA  4312, 174, 21, 96, 201, 77, 208, 4, 32, 5125
 2006 DATA  4320, 180, 23, 96, 201, 63, 208, 13, 173, 5277
 2007 DATA  4328, 0, 20, 208, 4, 32, 9, 25, 96, 4722
 2008 DATA  4336, 32, 38, 25, 96, 201, 84, 208, 4, 5024
 2009 DATA  4344, 32, 2, 30, 96, 96, 0, 0, 0, 4600
 2010 END
```

Appendix E12:

System Data Block for the Ohio Scientific C-1P

```
APPENDIX E12      SYSTEM DATA BLOCK FOR OSI C1P


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4096 TO 4119
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


2100 DATA  4096, 101, 208, 32, 24, 24, 211, 32, 16, 4744
2101 DATA  4104, 237, 254, 45, 191, 177, 252, 16, 16, 5292
2102 DATA  4112, 96, 96, 0, 0, 0, 0, 0, 0, 4304
2103 END

OK
```

Appendix E13:

System Data Block for the PET 2001

```
APPENDIX E13     SYSTEM DATA BLOCK FOR THE PET 2001


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4096 TO 4151
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


 2100 DATA  4096, 0, 128, 40, 39, 24, 131, 32, 30, 4520
 2101 DATA  4104, 42, 16, 210, 255, 16, 16, 16, 16, 4691
 2102 DATA  4112, 96, 41, 127, 56, 201, 64, 144, 17, 4858
 2103 DATA  4120, 201, 96, 144, 10, 162, 14, 141, 76, 4964
 2104 DATA  4128, 232, 233, 32, 24, 144, 3, 56, 233, 5085
 2105 DATA  4136, 64, 96, 32, 228, 255, 41, 127, 240, 5219
 2106 DATA  4144, 249, 96, 0, 0, 0, 0, 0, 0, 4489
 2107 END

OK
```

Appendix E14:

System Data Block for the Apple II

```
APPENDIX E14     SYSTEM DATA BLOCK FOR THE APPLE II


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  4096 TO 4127
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


2100 DATA  4096, 0, 4, 128, 39, 7, 7, 160, 222, 4663
2101 DATA  4104, 20, 16, 26, 16, 16, 16, 16, 16, 4246
2102 DATA  4112, 96, 9, 128, 96, 32, 12, 253, 41, 4779
2103 DATA  4120, 127, 96, 9, 128, 32, 253, 251, 96, 5112
2104 END

OK
```

Appendix E15:

System Data Block for the Atari 800

```
APPENDIX E15     SYSTEM DATA BLOCK FOR THE ATARI 800


THE FOLLOWING DATA STATEMENTS
CONTAIN DECIMAL OBJECT CODE AND
CHECKSUMS FOR MEMORY FROM  3712 TO 4223
SUITABLE FOR LOADING WITH
THE BASIC OBJECT CODE LOADER.


2100 DATA  3712, 32, 196, 17, 173, 179, 23, 72, 173, 4577
2101 DATA  3720, 178, 23, 72, 173, 85, 21, 72, 173, 4517
2102 DATA  3728, 84, 21, 72, 173, 83, 21, 72, 173, 4427
2103 DATA  3736, 82, 21, 72, 32, 43, 17, 165, 0, 4168
2104 DATA  3744, 141, 178, 23, 165, 1, 141, 179, 23, 4595
2105 DATA  3752, 32, 118, 17, 165, 0, 141, 82, 21, 4328
2106 DATA  3760, 165, 1, 141, 83, 21, 174, 3, 16, 4364
2107 DATA  3768, 172, 4, 16, 32, 60, 17, 165, 0, 4234
2108 DATA  3776, 141, 84, 21, 165, 1, 141, 85, 21, 4435
2109 DATA  3784, 32, 214, 23, 172, 4, 16, 162, 0, 4407
2110 DATA  3792, 32, 60, 17, 174, 3, 16, 160, 1, 4255
2111 DATA  3800, 32, 19, 17, 104, 141, 82, 21, 104, 4320
2112 DATA  3808, 141, 83, 21, 104, 141, 84, 21, 104, 4507
2113 DATA  3816, 141, 85, 21, 104, 141, 178, 23, 104, 4613
2114 DATA  3824, 141, 179, 23, 32, 211, 17, 96, 0, 4523
2115 DATA  3832, 0, 0, 0, 0, 0, 0, 0, 0, 3832
2116 DATA  3840, 108, 106, 59, 0, 0, 107, 43, 42, 4305
2117 DATA  3848, 111, 0, 112, 117, 13, 105, 45, 61, 4412
2118 DATA  3856, 118, 0, 99, 0, 0, 98, 120, 122, 4413
2119 DATA  3864, 52, 0, 51, 54, 27, 53, 50, 49, 4200
2120 DATA  3872, 44, 32, 46, 110, 0, 109, 47, 0, 4260
2121 DATA  3880, 114, 0, 101, 121, 9, 116, 119, 113, 4573
2122 DATA  3888, 57, 0, 48, 55, 8, 56, 60, 62, 4234
2123 DATA  3896, 102, 104, 100, 0, 0, 103, 115, 97, 4517
2124 DATA  3904, 76, 74, 58, 0, 0, 75, 91, 94, 4372
2125 DATA  3912, 79, 0, 80, 85, 13, 73, 45, 61, 4348
2126 DATA  3920, 86, 0, 67, 0, 0, 66, 88, 90, 4317
2127 DATA  3928, 52, 0, 51, 54, 27, 37, 34, 33, 4216
2128 DATA  3936, 90, 32, 93, 78, 0, 77, 63, 0, 4369
```

```
2129 DATA  3944, 82, 0, 69, 89, 9, 84, 87, 81, 4445
2130 DATA  3952, 40, 0, 41, 39, 127, 64, 0, 0, 4263
2131 DATA  3960, 70, 72, 68, 0, 0, 71, 83, 65, 4389
2132 DATA  3968, 0, 0, 0, 0, 0, 0, 0, 0, 3968
2133 DATA  3976, 0, 0, 16, 0, 0, 0, 0, 0, 3992
2134 DATA  3984, 0, 0, 3, 0, 0, 0, 0, 0, 3987
2135 DATA  3992, 0, 0, 0, 0, 0, 0, 0, 0, 3992
2136 DATA  4000, 0, 0, 0, 0, 0, 0, 0, 0, 4000
2137 DATA  4008, 0, 0, 0, 0, 0, 0, 0, 0, 4008
2138 DATA  4016, 0, 0, 0, 0, 0, 0, 0, 0, 4016
2139 DATA  4024, 6, 0, 0, 0, 0, 0, 0, 0, 4030
2140 DATA  4032, 0, 0, 0, 0, 0, 0, 0, 0, 4032
2141 DATA  4040, 0, 0, 0, 0, 0, 0, 0, 0, 4040
2142 DATA  4048, 0, 0, 0, 0, 0, 0, 0, 0, 4048
2143 DATA  4056, 0, 0, 0, 0, 0, 0, 0, 0, 4056
2144 DATA  4064, 0, 0, 0, 0, 0, 0, 0, 0, 4064
2145 DATA  4072, 0, 0, 0, 0, 0, 0, 0, 0, 4064
2146 DATA  4080, 0, 0, 0, 0, 0, 0, 0, 0, 4072
2147 DATA  4088, 0, 0, 0, 0, 0, 0, 0, 0, 4080
2148 DATA  4096, 66, 124, 40, 39, 23, 127, 0, 123, 4638
2149 DATA  4104, 40, 16, 54, 16, 16, 16, 16, 16, 4294
2150 DATA  4112, 96, 41, 127, 56, 201, 32, 144, 8, 4817
2151 DATA  4120, 201, 96, 144, 8, 201, 123, 144, 7, 5044
2152 DATA  4128, 173, 6, 16, 96, 56, 233, 32, 96, 4836
2153 DATA  4136, 173, 252, 2, 201, 255, 240, 249, 168, 5676
2154 DATA  4144, 185, 0, 15, 96, 0, 0, 201, 13, 4654
2155 DATA  4152, 208, 6, 169, 0, 141, 53, 16, 96, 4841
2156 DATA  4160, 201, 10, 208, 3, 76, 128, 14, 141, 4941
2157 DATA  4168, 52, 16, 32, 196, 17, 172, 4, 16, 4673
2158 DATA  4176, 174, 53, 16, 32, 60, 17, 173, 52, 4753
2159 DATA  4184, 16, 32, 124, 17, 238, 53, 16, 173, 4853
2160 DATA  4192, 53, 16, 205, 3, 16, 208, 6, 32, 4731
2161 DATA  4200, 58, 16, 32, 128, 14, 32, 211, 17, 4708
2162 DATA  4208, 96, 0, 0, 0, 0, 0, 0, 0, 4304
2163 DATA  4216, 0, 0, 0, 0, 0, 0, 0, 0, 4216
2164 END

OK
```

Index

Software

Beyond Games: Systems Software for Your 6502 Personal Computer

By Ken Skier

Use your 6502 personal computer for more than games! Learn how it works and how to make it work for you. This book, for Apple, Atari, Ohio Scientific and PET computer owners who know little or nothing about bits, bytes, hardware, and software, presents a guided tour of your computer. Beginning with basic concepts such as *what is memory?* and *what is a program?*, **Beyond Games** moves through a fast but surprisingly complete course in assembly language programming. Having mastered these fundamentals, the reader is introduced to many useful subroutines and programming tools, such as screen utilities, print utilities, a machine language monitor, a hexadecimal dump tool, a move tool, a disassembler, and a simple, screen-based text editor.

About the Author

Ken Skier, systems analyst for Wang Laboratories, Inc, designs software for word processing and other applications concerning the office of the future. A Massachusetts Institute of Technology graduate, he co-founded the M.I.T. Writing Program, where he teaches science fiction writing. He lives in Cambridge, Massachusetts, with his wife Cynthia and a nameless white cat.

0-07-057860-5